# Dynamic Indexes

## Advanced Programming and Algorithmic Design

Alberto Casagrande
*Email:* acasagrande@units.it

a.a. 2018/2019

# Motivations

# A Simple Problem for Registry Office

Let us consider the registry office

For each newborn, they record a set of data e.g., name, birthday, parents, etc.

So, the registry data-set (hopefully) changes quite often

What if they frequently perform a birthday-based search on the data-set? E.g., Find all the baby born a given (variable) day?

# Some Possible Strategies

They may:

# Some Possible Strategies

They may:

- scan all the data-set at each query

Takes time $\Theta(n)$

# Some Possible Strategies

They may:

- scan all the data-set at each query

  Takes time $\Theta(n)$

- sort the data-set by birthday at each insertion

  Takes time $\Theta(n)$ per insertion (Radix Sort) + $\Theta(\log n)$ per query. What if they also want to perform searches by name?

# Some Possible Strategies

They may:

- scan all the data-set at each query

  Takes time $\Theta(n)$

- sort the data-set by birthday at each insertion

  Takes time $\Theta(n)$ per insertion (Radix Sort) + $\Theta(\log n)$ per query. What if they also want to perform searches by name?

- sort the data-set by birthday at each query

  Takes time $\Theta(n + \log n) = \Theta(n)$ per query (Radix Sort and DS)

## Some Possible Strategies

They may:

- scan all the data-set at each query

  Takes time $\Theta(n)$

- sort the data-set by birthday at each insertion

  Takes time $\Theta(n)$ per insertion (Radix Sort) + $\Theta(\log n)$ per query. What if they also want to perform searches by name?

- sort the data-set by birthday at each query

  Takes time $\Theta(n + \log n) = \Theta(n)$ per query (Radix Sort and DS)

The data-set often changes, thus, array is not the most suitable data structure to achieve this goal

# A Dynamic Data Structure for Indexing

We need a data structure providing (efficient) support for:

- adding new data
- searching data
- removing data

# A Dynamic Data Structure for Indexing

We need a data structure providing (efficient) support for:

- adding new data
- searching data
- removing data

We are aiming to build an index i.e., an auxiliary data structure to "efficiently" perform above operations

# Binary Search Trees

# Binary Search Tree Notation

We have already introduced trees as ADT.

As far as searching trees concern, the following notation holds

# Binary Search Tree Notation

We have already introduced trees as ADT.

As far as searching trees concern, the following notation holds

`x.left` and `x.right` are the left and right children of $x$, respectively

# Binary Search Tree Notation

We have already introduced trees as ADT.

As far as searching trees concern, the following notation holds

`x.left` and `x.right` are the left and right children of $x$, respectively

`x.parent` is the parent of $x$

# Binary Search Tree Notation

We have already introduced trees as ADT.

As far as searching trees concern, the following notation holds

`x.left` and `x.right` are the left and right children of $x$, respectively

`x.parent` is the parent of $x$

`x.key` is the value stored in $x$ i.e., its key

# Binary Search Tree Notation

We have already introduced trees as ADT.

As far as searching trees concern, the following notation holds

`x.left` and `x.right` are the left and right children of $x$, respectively

`x.parent` is the parent of $x$

`x.key` is the value stored in $x$ i.e., its key

`x.left=NIL`, then $x$ misses the left child

## Binary Search Tree Notation

We have already introduced trees as ADT.

As far as searching trees concern, the following notation holds

`x.left` and `x.right` are the left and right children of $x$, respectively

`x.parent` is the parent of $x$

`x.key` is the value stored in $x$ i.e., its key

`x.left=NIL`, then $x$ misses the left child

`T.root` is the root of the tree $T$ and `T.root.parent=NIL`

# Some Useful Θ(1) Functions

```
def IS_RIGHT_CHILD(x):
  return x.parent≠NIL and x.parent.right=x
enddef

def SIBLING(x):      # get x's sibling
  if IS_RIGHT_CHILD(x):
    return x.left
  endif
  return x.right
enddef

def UNCLE(x):       #get x's uncle
  return SIBLING(x.parent)
enddef
```

# Some Useful $\Theta(1)$ Functions (Cont'd)

```
def CHILDHOOD_SIDE(x): # get x's side w.r.t.
                       # its parent
  if IS_RIGHT_CHILD(x):
    return RIGHT
  endif
  return LEFT
enddef

def REVERSE_SIDE(side):   # reverse the side
  if side = LEFT:
    return RIGHT
  endif
  return LEFT
enddef
```

## Some Useful $\Theta(1)$ Functions (Cont'd 2)

```
def GET_CHILD(x, side):  # get x's child on side
  if side = LEFT:
    return x.left
  endif

  return x.right
enddef

def SET_CHILD(x, side, y):  # set x's child
  if side = LEFT:
    x.left ← y
  else:
    x.right ← y
  endif
enddef
```

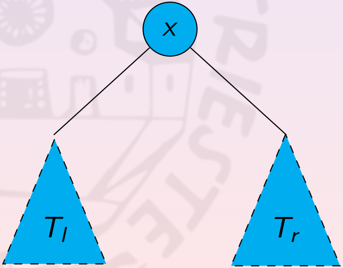
# Some Useful $\Theta(1)$ Functions (Cont'd 3)

```
def GRANDPARENT(x):        # get x's granpa
  return x.parent.parent
enddef
```

# Binary Search Trees

A Binary Search Tree (BST) is a tree s.t.:

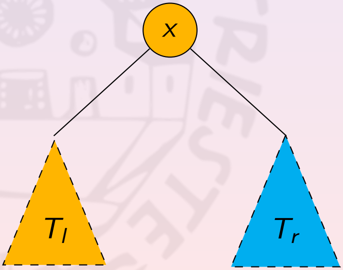
- all the keys belong to a totally ordered set w.r.t. $\preceq$



# Binary Search Trees

A Binary Search Tree (BST) is a tree s.t.:

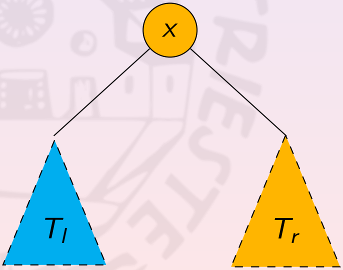
- all the keys belong to a totally ordered set w.r.t. $\preceq$
- if $x_l$ is in the left sub-tree of $x$, then $x_l.key \preceq x.key$



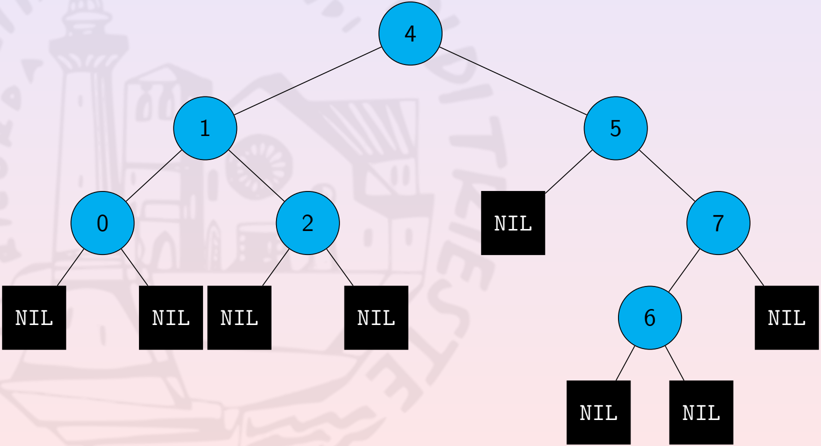
# Binary Search Trees

A Binary Search Tree (BST) is a tree s.t.:

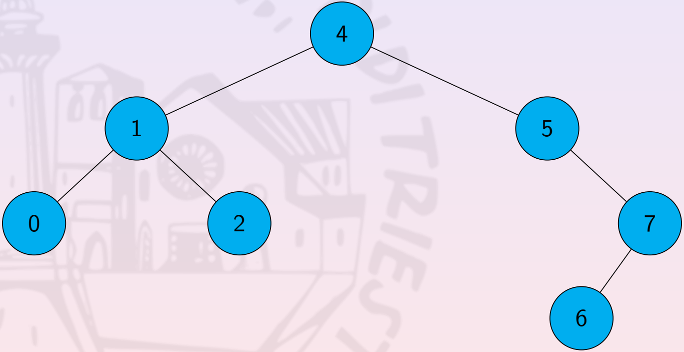
- all the keys belong to a totally ordered set w.r.t. $\preceq$
- if $x_l$ is in the left sub-tree of $x$, then $x_l.key \preceq x.key$
- if $x_r$ is in the right sub-tree of $x$, then $x.key \preceq x_r.key$



# Binary Search Trees: an Example



# Binary Search Trees: an Example



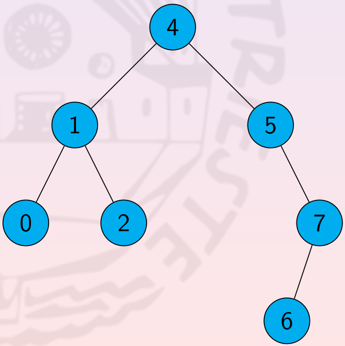
# In-Order Walk

```
def INORDER_WALK_AUX(x):
  if x ≠ NIL:
    INORDER_WALK_AUX(x.left)
    print x.key
    INORDER_WALK_AUX(x.right)
  endif
endif

def INORDER_WALK(T):
  INORDER_WALK_AUX(T.root)
endif
```

# Searching for the Maximum/Minimum

Due to the BST property:

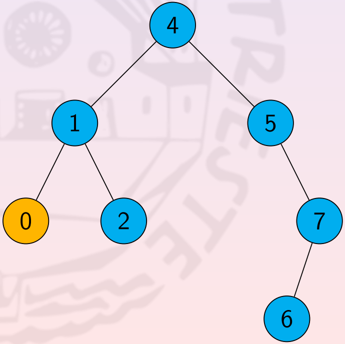
- the minimum key is contained by the first node on the leftmost branch that has not a left child



# Searching for the Maximum/Minimum

Due to the BST property:

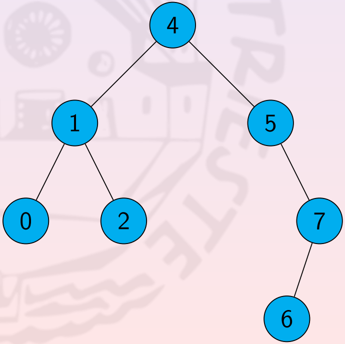
- the minimum key is contained by the first node on the leftmost branch that has not a left child



# Searching for the Maximum/Minimum

Due to the BST property:

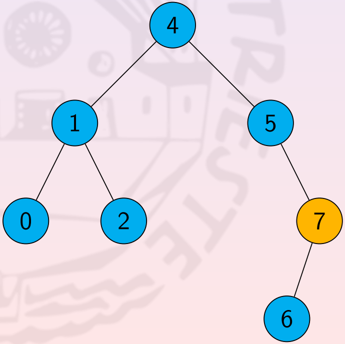
- the minimum key is contained by the first node on the leftmost branch that has not a left child
- the maximum key is contained by the first node on the rightmost branch that has not a right child



# Searching for the Maximum/Minimum

Due to the BST property:

- the minimum key is contained by the first node on the leftmost branch that has not a left child
- the maximum key is contained by the first node on the rightmost branch that has not a right child



## Searching for the Maximum/Minimum: Pseudo-Code

```
def MINIMUM_IN_TREE(x):
  while x.left ≠ NIL:
    x ← x.left
  endif
  return x
endif


def MAXIMUM_TREE(x):
  while x.right ≠ NIL:
    x ← x.right
  endif
  return x
endif
```

# Searching for the Maximum/Minimum: Pseudo-Code

```
def MINIMUM_IN_TREE(x):
  while x.left ≠ NIL:
    x ← x.left
  endif
  return x
endif
```

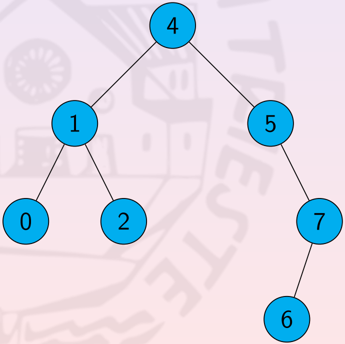
$O(h_T)$

```
def MAXIMUM_TREE(x):
  while x.right ≠ NIL:
    x ← x.right
  endif
  return x
endif
```

$O(h_T)$

# Successor of a Node

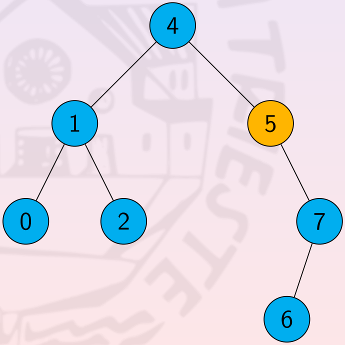
Due to the BST property, the successor w.r.t. $\preceq$ of $n$ is either



# Successor of a Node

Due to the BST property, the successor w.r.t. $\preceq$ of $n$ is either

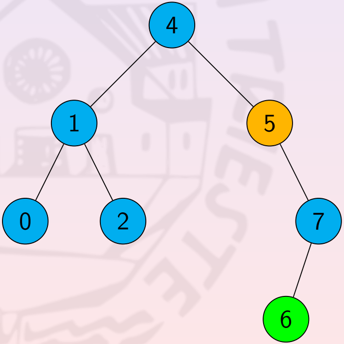
- the node containing the minimum in the left sub-tree of $n$



# Successor of a Node

Due to the BST property, the successor w.r.t. $\preceq$ of $n$ is either

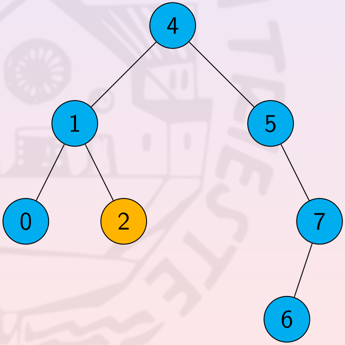
- the node containing the minimum in the left sub-tree of $n$ or



# Successor of a Node

Due to the BST property, the successor w.r.t. $\preceq$ of $n$ is either

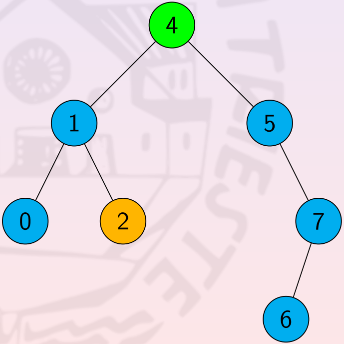
- the node containing the minimum in the left sub-tree of $n$ or
- the nearest "left-ancestor" of $n$, if $n$ has not left child



# Successor of a Node

Due to the BST property, the successor w.r.t. $\preceq$ of $n$ is either

- the node containing the minimum in the left sub-tree of $n$ or
- the nearest "left-ancestor" of $n$, if $n$ has not left child



## Successor of a Node: Pseudo-Code

```
def SUCCESSOR(x):
  if x.left ≠ NIL:
    return SEARCH_MIN_SUBTREE(x)
  endif

  y ← x.parent
  while y ≠ NIL and IS_RIGHT_CHILD(x):
    x ← y
    y ← x.parent
  endwhile

  return y
enddef
```

# Successor of a Node: Pseudo-Code

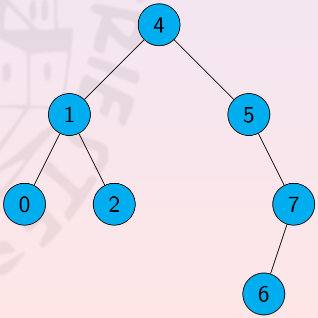
```
def SUCCESSOR(x):
  if x.left ≠ NIL:
    return SEARCH_MIN_SUBTREE(x)
  endif

  y ← x.parent
  while y ≠ NIL and IS_RIGHT_CHILD(x):    O(h_T)
    x ← y
    y ← x.parent
  endwhile

  return y
enddef
```

# Searching for a Value in a BST

To search for $v$, apply a dichotomic approach from the root $n$:

- if $n$ is `NIL` or $n.key = v$, return $n$
- if $n.key \preceq v$, search on the right sub-tree
- if $n.key \not\preceq v$, search on the left sub-tree

Searching for 3

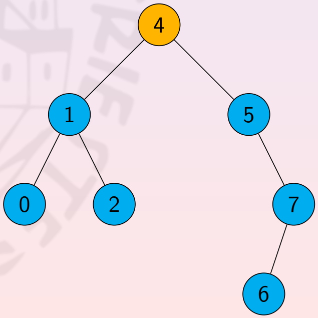


# Searching for a Value in a BST

To search for $v$, apply a dichotomic approach from the root $n$:

- if $n$ is `NIL` or $n.key = v$, return $n$
- if $n.key \preceq v$, search on the right sub-tree
- **if $n.key \not\preceq v$, search on the left sub-tree**

Searching for 3

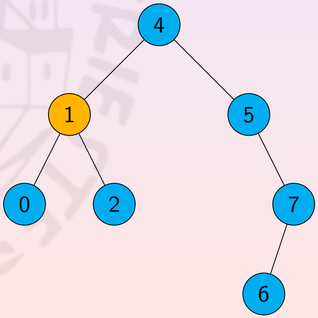


# Searching for a Value in a BST

To search for $v$, apply a dichotomic approach from the root $n$:

- if $n$ is `NIL` or $n.key = v$, return $n$
- **if $n.key \preceq v$, search on the right sub-tree**
- if $n.key \not\preceq v$, search on the left sub-tree

Searching for 3

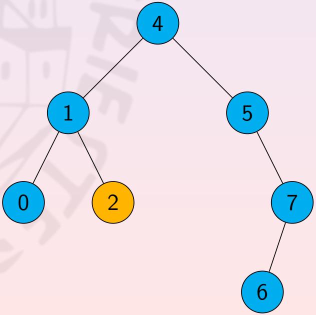


# Searching for a Value in a BST

To search for $v$, apply a dichotomic approach from the root $n$:

- if $n$ is $\mathtt{NIL}$ or $n.key = v$, return $n$
- **if $n.key \preceq v$, search on the right sub-tree**
- if $n.key \not\preceq v$, search on the left sub-tree

Searching for 3

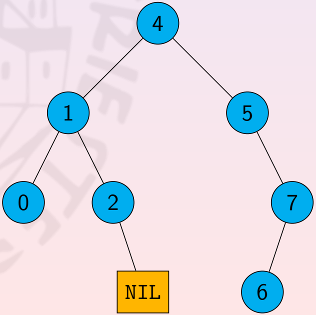


# Searching for a Value in a BST

To search for $v$, apply a dichotomic approach from the root $n$:

- **if** $n$ **is** `NIL` **or** $n.key = v$**, return** $n$
- if $n.key \preceq v$, search on the right sub-tree
- if $n.key \not\preceq v$, search on the left sub-tree

Searching for 3



## Searching for a Value in a BST: Pseudo-Code

```
def SEARCH_SUBTREE(x, v):
  if x = NIL:
    return x
  endif

  if x.key ≼ v:
    if v ≼ x.key:
      return x
    endif

    return SEARCH_SUBTREE(x.right, v)
  endif

  return SEARCH_SUBTREE(x.left, v)
enddef
```

# Searching for a Value in a BST: Complexity

At each iteration, algorithm performs $\Theta(1)$ operations

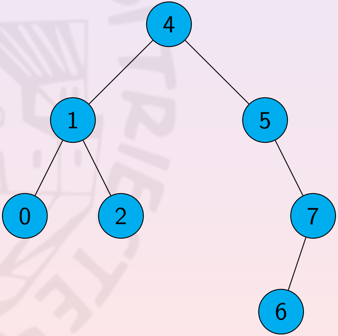
The # of iterations depends on the height $h_T$ of $T$ and on $v$

The algorithm takes time $O(h_T)$

# Inserting a Value in a BST

Browse a branch of $T$ and add a new leaf having $v$ as key

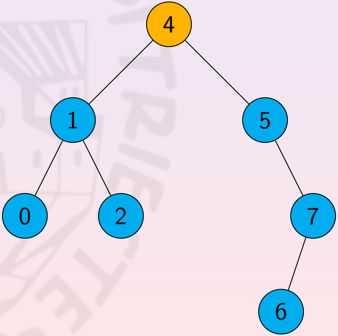
Inserting 3



# Inserting a Value in a BST

Browse a branch of $T$ and add a new leaf having $v$ as key

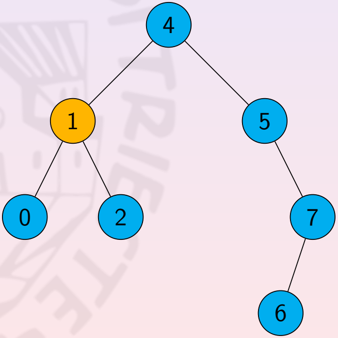
Inserting 3



# Inserting a Value in a BST

Browse a branch of $T$ and add a new leaf having $v$ as key

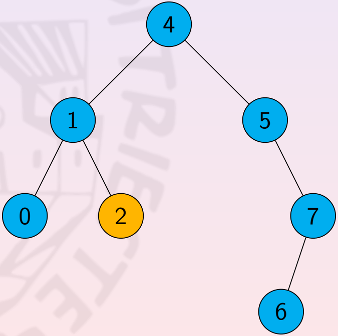
Inserting 3



# Inserting a Value in a BST

Browse a branch of $T$ and add a new leaf having $v$ as key

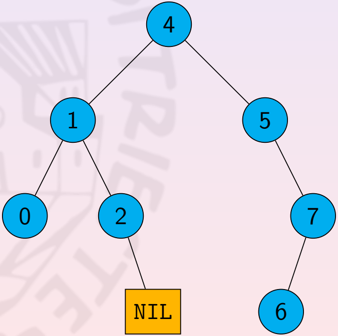
Inserting 3



# Inserting a Value in a BST

Browse a branch of $T$ and add a new leaf having $v$ as key

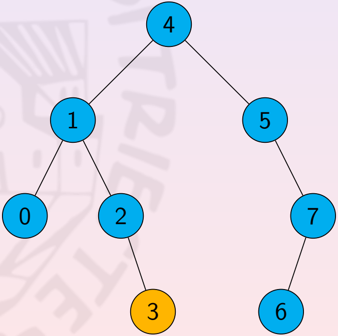
Inserting 3



# Inserting a Value in a BST

Browse a branch of $T$ and add a new leaf having $v$ as key

Inserting 3



## Inserting a Value in a BST: Pseudo-Code

```
def INSERT_BST(T,z): # z is the new node
  x ← T.root
  y ← NIL    # y is x's parent

  # search the right place for z
  while x≠NIL:
    y ← x
    if z.key ⪯ x.key:
      x ← x.left
    else:
      x ← x.right
    endif
  endwhile
```

# Inserting a Value in a BST: Pseudo-Code

```
def INSERT_BST(T,z): # z is the new node
  x ← T.root
  y ← NIL     # y is x's parent

  # search the right place for z
  while x≠NIL:
    y ← x
    if z.key ⪯ x.key:
      x ← x.left
    else:
      x ← x.right
    endif
  endwhile
```

$O(h_T)$

# Inserting a Value in a BST: Pseudo-Code (Cont'd)

```
  # attaching the new node

  z.parent ← y
  if y=NIL:
    T.root ← z
  else:
    if z.key ⪯ y.key:
      y.left ← z
    else:
      y.right ← z
    endif
  endif
enddef
```

# Inserting a Value in a BST: Pseudo-Code (Cont'd)

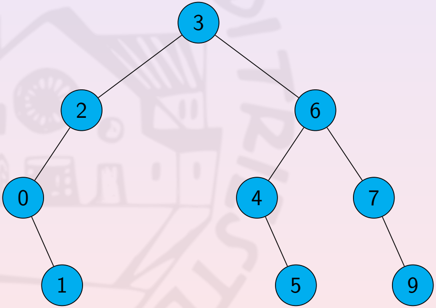
```
# attaching the new node

z.parent ← y
if y=NIL:
  T.root ← z
else:
  if z.key ⪯ y.key:
    y.left ← z
  else:
    y.right ← z
  endif
endif
enddef
```

$\Theta(1)$

# Removing a Key from a BST

We want to remove $z$. Either

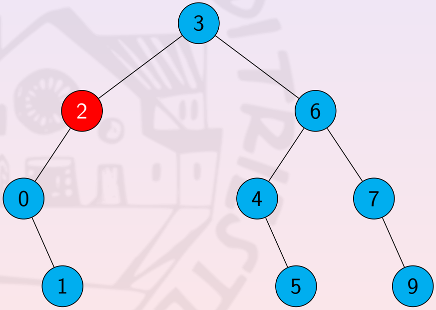
- $z$ has one child at most



# Removing a Key from a BST

We want to remove $z$. Either

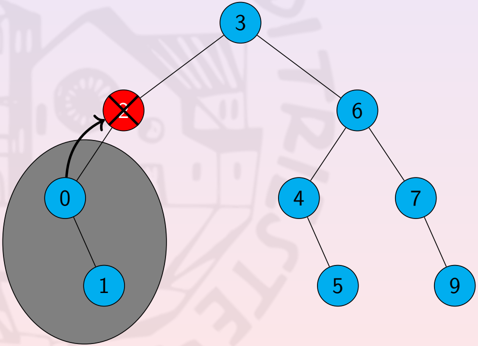
- $z$ has one child at most



# Removing a Key from a BST

We want to remove $z$. Either

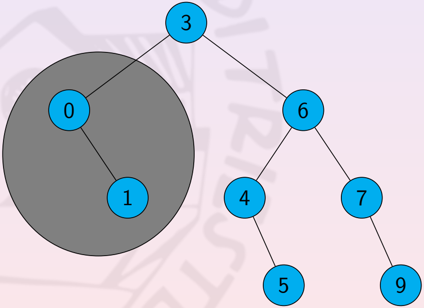
- $z$ has one child at most



# Removing a Key from a BST

We want to remove $z$. Either

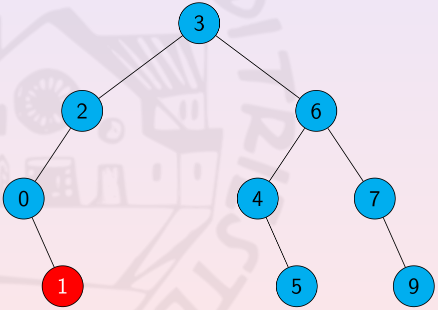
- $z$ has one child at most



# Removing a Key from a BST

We want to remove $z$. Either

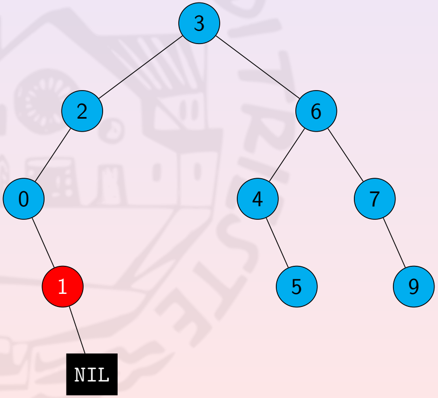
- $z$ has one child at most



# Removing a Key from a BST

We want to remove $z$. Either

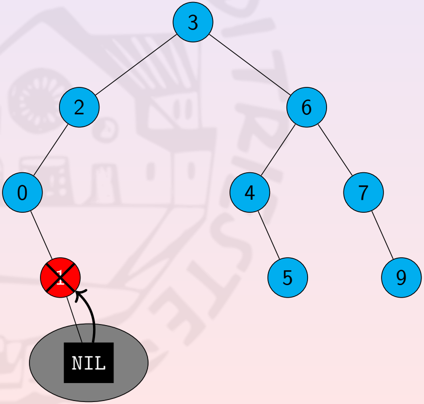
- $z$ has one child at most



# Removing a Key from a BST

We want to remove $z$. Either

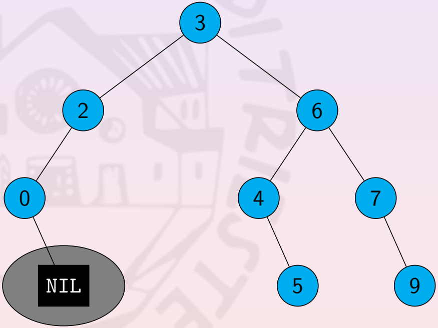
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

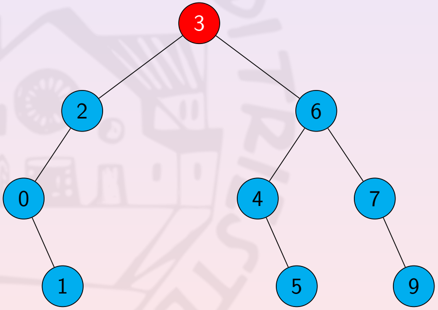
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

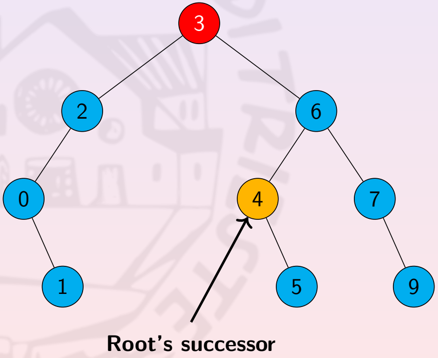
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

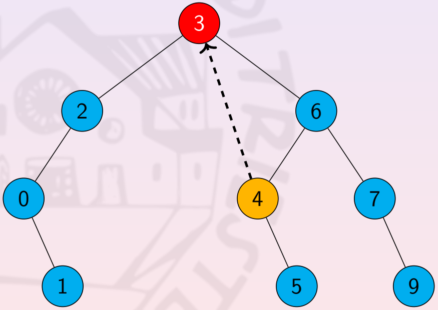
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

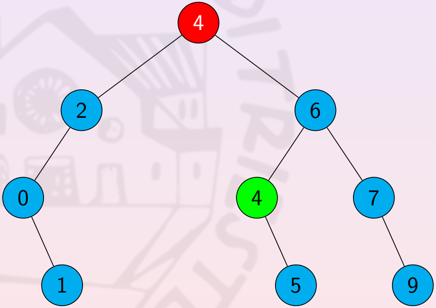
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

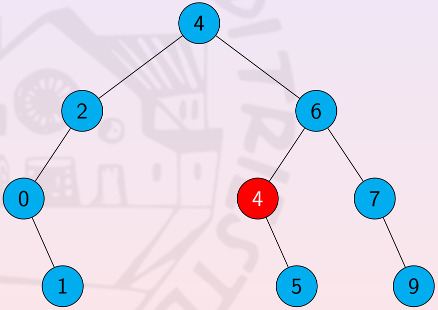
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

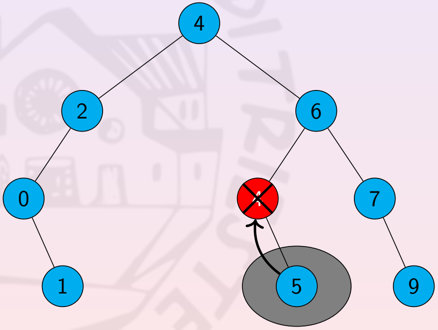
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

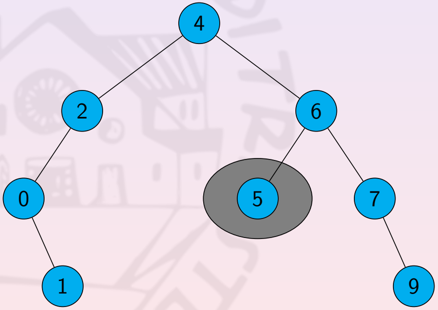
- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST

We want to remove $z$. Either

- $z$ has one child at most or
- $z$ has two children



# Removing a Key from a BST: Pseudo-Code

```
def TRANSPLANT(T,x,y): # replace x by y
  if x.parent=NIL: # x is T's root
    T.root ← y
  else:                # x has a parent
    x_side ← CHILDHOOD_SIDE(x)
    GET_CHILD(x.parent, x_side) ← y    # attach y in place of x
  endif

  if y≠NIL:          # update y's parent
    y.parent ← x.parent
  endif
enddef
```

# Removing a Key from a BST: Pseudo-Code (Cont'd)

```
def REMOVE_BST(T,z): # remove z.key from T and
                      # return a removed node
  if z.left=NIL:        # if z lacks of left child
    TRANSPLANT(T,z,z.right)
    return z
  endif

  if z.right=NIL:       # if z lacks of right child
    TRANSPLANT(T,z,z.left)
    return z
  endif

  y ← MINIMUM_IN_TREE(z.right)
  z.key ← y.key

  return REMOVE_BST(T,y) # y lacks of left child
enddef
```

# Remove a Node from a BST: Complexity

`TRANSPLANT` costs $\Theta(1)$, while `MINIMUM_IN_TREE` $O(h_T)$

So, if $z$ has at most one child, removing it costs $\Theta(1)$

In the general case, removing $z$ takes time $O(h_T)$

# To Summarize BSTs. . .

| | |
|---|---|
| **Creation** | $\Theta(1)$ |
| **Insertion** | $O(h_T)$ |
| **Search** | $O(h_T)$ |
| **Deletion** | $O(h_T)$ |

# To Summarize BSTs...

| | |
|---|---|
| **Creation** | $\Theta(1)$ |
| **Insertion** | $O(h_T)$ |
| **Search** | $O(h_T)$ |
| **Deletion** | $O(h_T)$ |

However, $h_T$ may be equal to the number $n$ of nodes e.g., keep inserting always the maximum

# To Summarize BSTs. . .

| | |
|---|---|
| **Creation** | $\Theta(1)$ |
| **Insertion** | $O(n)$ |
| **Search** | $O(n)$ |
| **Deletion** | $O(n)$ |

However, $h_T$ may be equal to the number $n$ of nodes e.g., keep inserting always the maximum

BSTs are exceeded by single-linked lists (insertion $\Theta(1)$)

# Balancing BSTs

The minimum height for a binary tree having $n$ nodes is $\lceil \log_2 n \rceil$

We aim to balance the trees we dealing with i.e., bring their heights to $O(\log n)$

How to do it?

# Balancing BSTs

The minimum height for a binary tree having $n$ nodes is $\lceil \log_2 n \rceil$

We aim to balance the trees we dealing with i.e., bring their heights to $O(\log n)$

How to do it?

GLOBALLY: balance the whole BST after each insertion/deletion

# Balancing BSTs

The minimum height for a binary tree having $n$ nodes is $\lceil \log_2 n \rceil$

We aim to balance the trees we dealing with i.e., bring their heights to $O(\log n)$

How to do it?

GLOBALLY: balance the whole BST after each insertion/deletion $O(n)$

# Balancing BSTs

The minimum height for a binary tree having $n$ nodes is $\lceil \log_2 n \rceil$

We aim to balance the trees we dealing with i.e., bring their heights to $O(\log n)$

How to do it?

GLOBALLY: balance the whole BST after each insertion/deletion $O(n)$

LOCALLY: balance only the "unbalanced" part of the tree

# Balancing BSTs

The minimum height for a binary tree having $n$ nodes is $\lceil \log_2 n \rceil$

We aim to balance the trees we dealing with i.e., bring their heights to $O(\log n)$

How to do it?

GLOBALLY: balance the whole BST after each insertion/deletion
$O(n)$

LOCALLY: balance only the "unbalanced" part of the tree
How to know if it is unbalanced?

# Balancing BSTs

The minimum height for a binary tree having $n$ nodes is $\lceil \log_2 n \rceil$

We aim to balance the trees we dealing with i.e., bring their heights to $O(\log n)$

How to do it?

GLOBALLY: balance the whole BST after each insertion/deletion $O(n)$

LOCALLY: balance only the "unbalanced" part of the tree How to know if it is unbalanced? How to handle branches' lengths?

# Red-Black Trees

# RBTs: Definition

Are BSTs satisfying the following conditions:

- each node is either a RED or a BLACK node
- the tree's root is BLACK
- all the leaves are BLACK `NIL` nodes
- all the RED nodes must have BLACK children
- for each node $n$, all the branches from $n$ contain the same # of black nodes

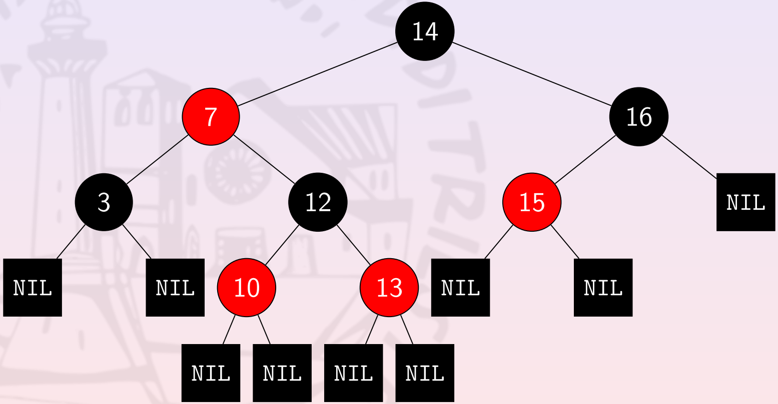
# RBTs: Definition

Are BSTs satisfying the following conditions:

- each node is either a RED or a BLACK node
- the tree's root is BLACK
- all the leaves are BLACK `NIL` nodes
- all the RED nodes must have BLACK children
- for each node $n$, all the branches from $n$ contain the same $\#$ of black nodes

$BH(x)$ will be the $\#$ of BLACK nodes $\neq x$ in any branch from $x$

# RBTs: An Example



# How “Tall” Are RB-Trees?

**Theorem (Heights of a RB-Tree)**

*Any RBT with n internal nodes has height at most* $2\log_2(n+1)$

Proof Sketch:

- prove that the sub-tree rooted in $x$ has at least $2^{BH(x)}-1$ internal nodes
- $BH(x)$ is at least half of $x$'s height $h$ then

$$n \geq 2^{h/2}-1$$

# How "Tall" Are RB-Trees?

The ratio between $x$'s height and $BH(x)$ is topped by 2

**Theorem (Heights of a RB-Tree)**

*Any RBT with n internal nodes has height at most* $2\log_2(n+1)$

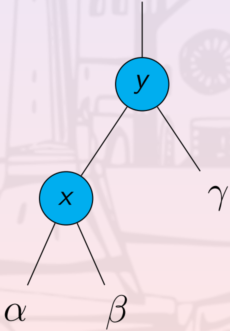
Proof Sketch:

- prove that the sub-tree rooted in $x$ has at least $2^{BH(x)}-1$ internal nodes
- $BH(x)$ is at least half of $x$'s height $h$ then

$$n \geq 2^{h/2}-1$$

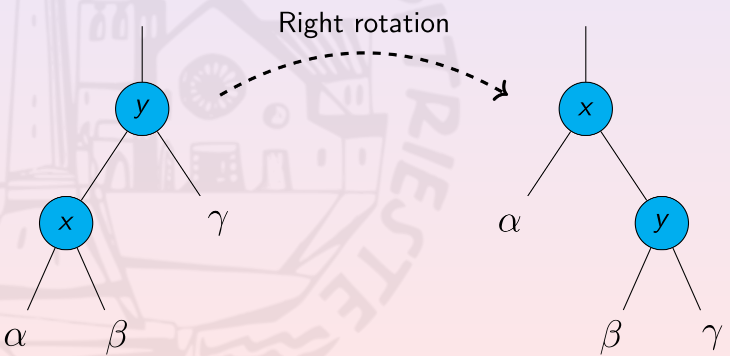
# Rotating a Sub-Tree

Rotations are operations on the tree structure preserving the binary searching tree property



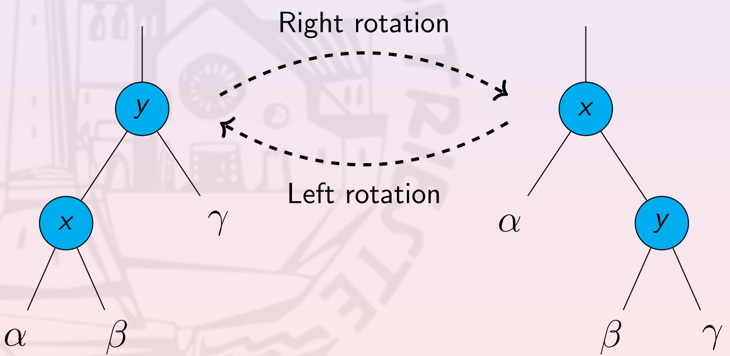
# Rotating a Sub-Tree

Rotations are operations on the tree structure preserving the binary searching tree property



# Rotating a Sub-Tree

Rotations are operations on the tree structure preserving the binary searching tree property



# Rotating a Sub-Tree: Pseudo-Code

```
def ROTATE(T,x,side):
  r_side ← REVERSE_SIDE(side)

  y ← GET_CHILD(x, r_side)
  TRANSPLANT(T,x,y)

  beta ← GET_CHILD(y, side)
  TRANSPLANT(T,beta,x)

  SET_CHILD(x,r_side,beta) # move beta
  if beta ≠ NIL:
    beta.parent ← x
  endif
enddef
```

## Rotating a Sub-Tree: Pseudo-Code

```
def ROTATE(T,x,side):
  r_side ← REVERSE_SIDE(side)

  y ← GET_CHILD(x, r_side)
  TRANSPLANT(T,x,y)

  beta ← GET_CHILD(y, side)
  TRANSPLANT(T,beta,x)

  SET_CHILD(x,r_side,beta) # move beta
  if beta ≠ NIL:
    beta.parent ← x
  endif
enddef
```

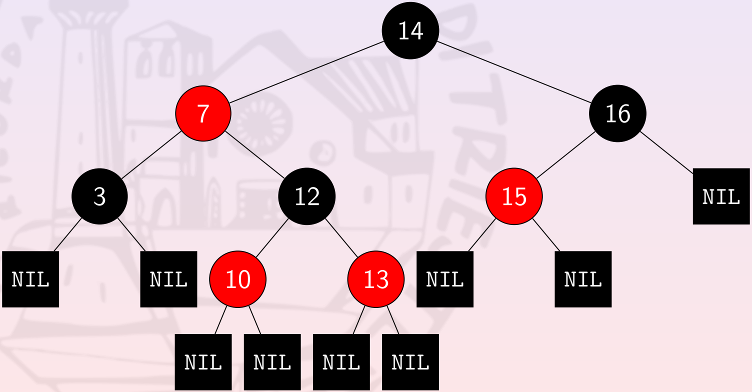
$\Theta(1)$

# Inserting a New Node

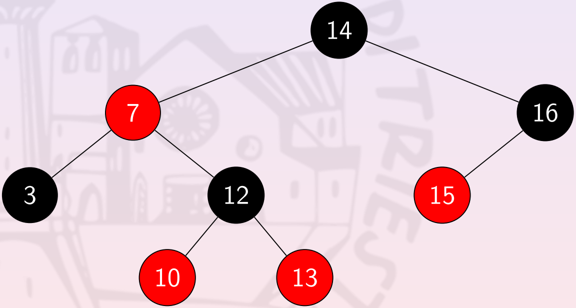
Requires:

- inserting as in BST
- RED coloring the node
- fixing up RB-Tree properties

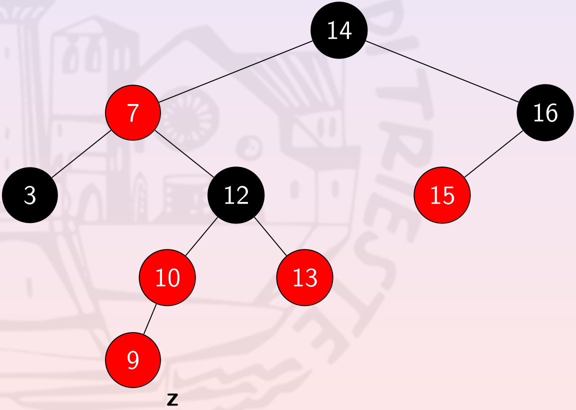
# Inserting a New Node: Example



# Inserting a New Node: Example

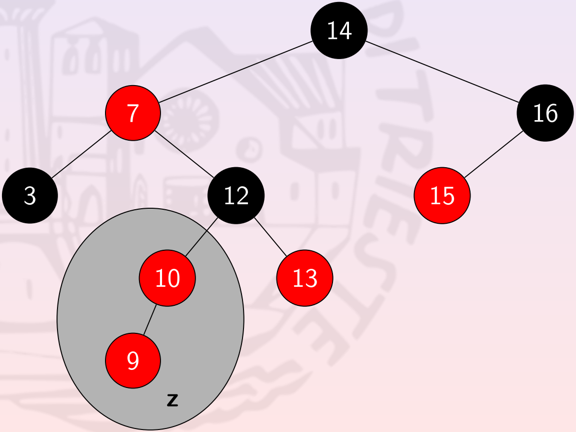


# Inserting a New Node: Example



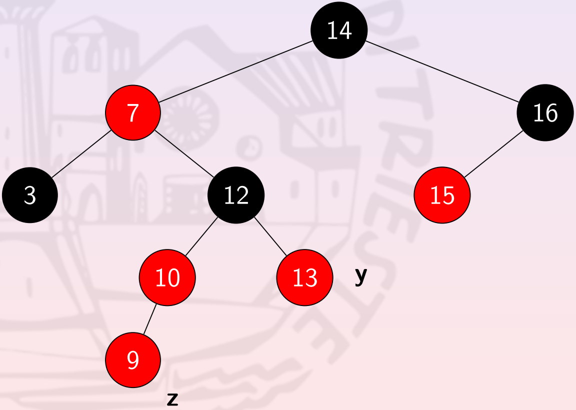
# Inserting a New Node: Example

$z$'s parent may be RED. How to fix it?



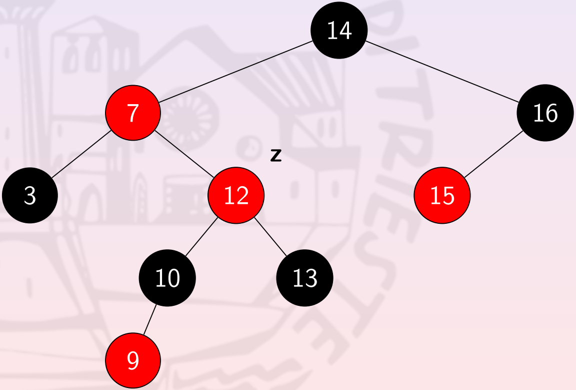
# Inserting a New Node: Example

**Case 1:** $z$'s uncle ($y$) is RED...



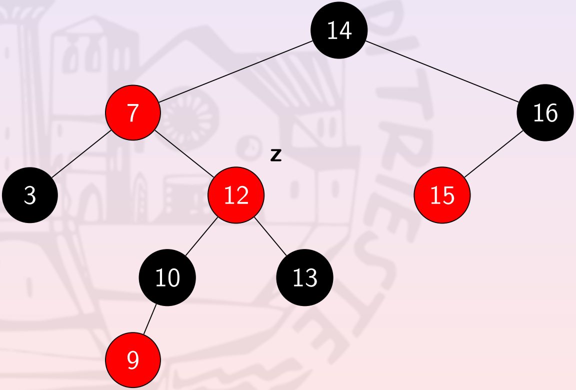
# Inserting a New Node: Example

**Case 1:** $z$'s uncle ($y$) is RED. . . RED color $z$'s granpa and BLACK color $z$'s parent and $y$.



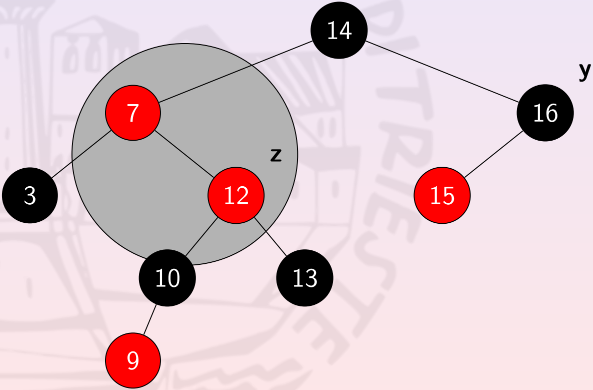
# Inserting a New Node: Example

**Case 1:** $z$'s uncle ($y$) is RED...



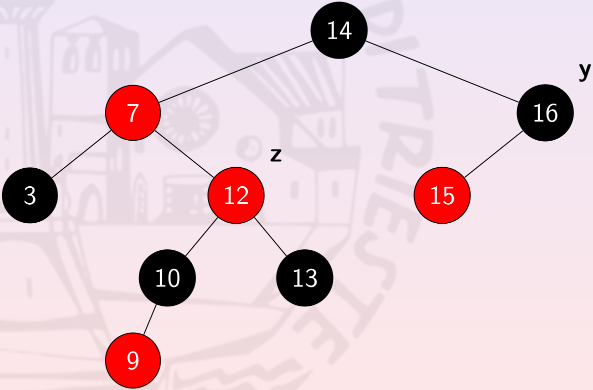
# Inserting a New Node: Example

Still facing problems and $z$'s uncle is BLACK (no Case 1)



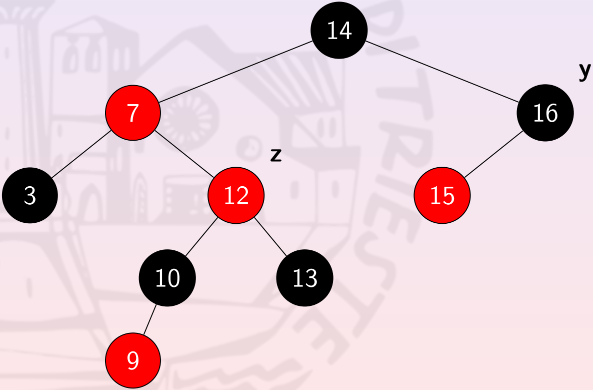
# Inserting a New Node: Example

**Case 2:** $y$ is BLACK and $y$ and $z$ are both right children.



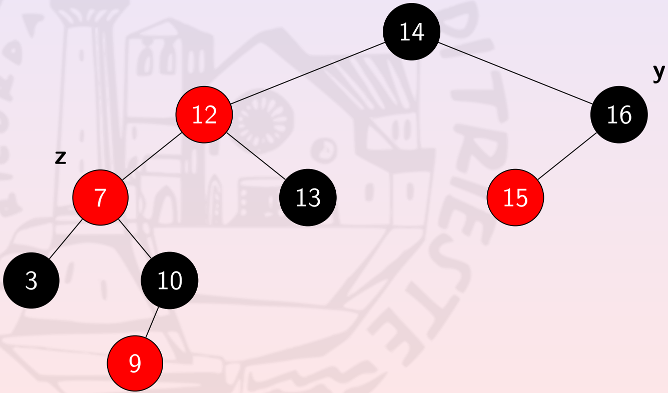
# Inserting a New Node: Example

**Case 2:** $y$ is BLACK and $y$ and $z$ are both right children. Rotate left on $z$'s parent.



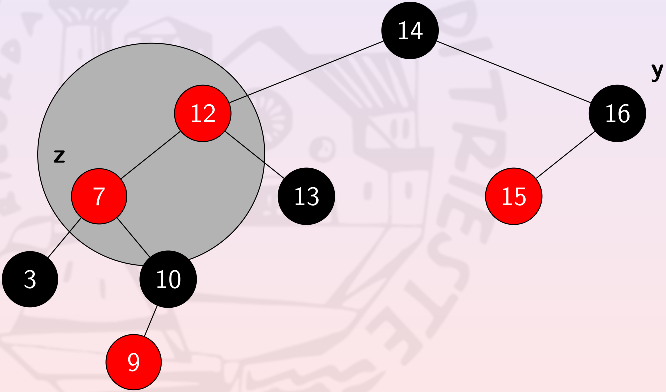
# Inserting a New Node: Example

**Case 2:** $y$ is BLACK and $y$ and $z$ are both right children. Rotate left on $z$'s parent. New $z$ is former $z$'s parent



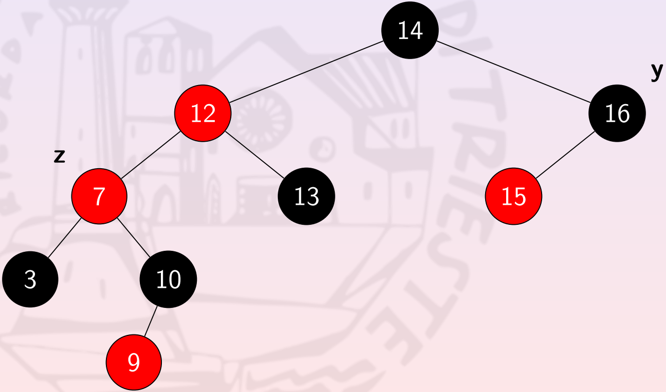
# Inserting a New Node: Example

Still facing problems, but



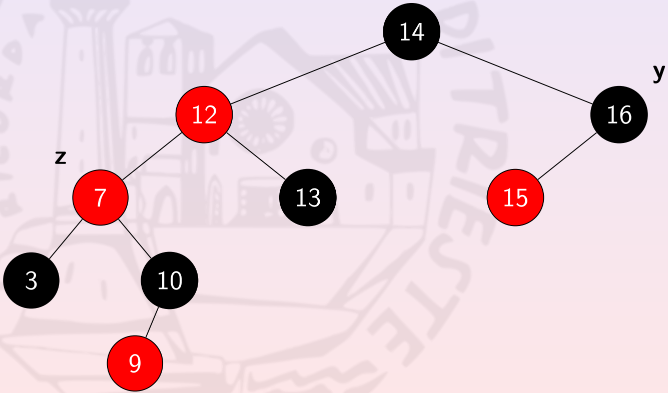
# Inserting a New Node: Example

Still facing problems, but $y$ is still BLACK (no Case 1)



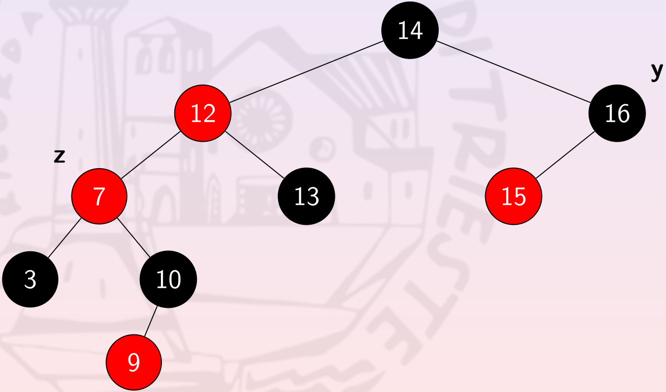
# Inserting a New Node: Example

Still facing problems, but $y$ is still BLACK (no Case 1) and $z$ and $y$ are on different sides w.r.t. their parents (no Case 2)



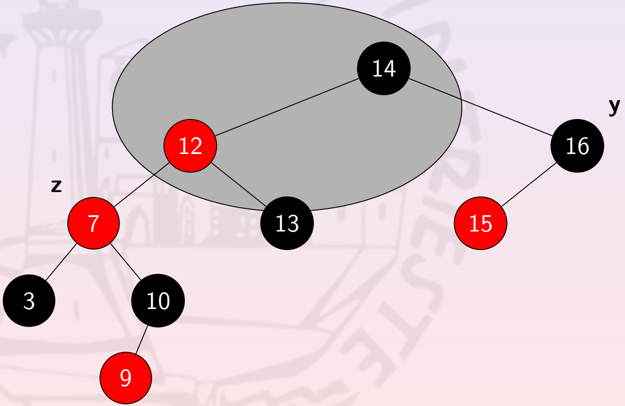
# Inserting a New Node: Example

**Case 3:** $y$ is BLACK and $y$ and $z$ are right and left children.



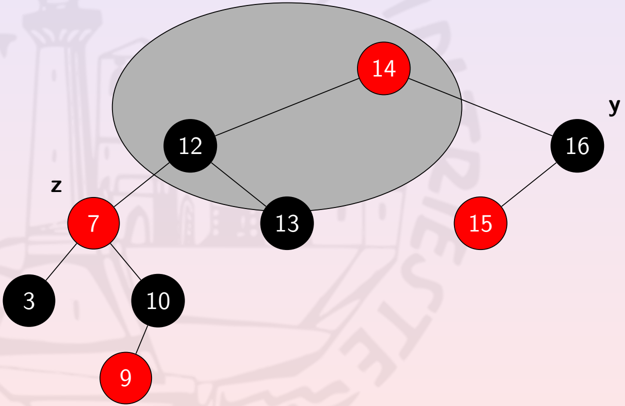
# Inserting a New Node: Example

**Case 3:** $y$ is BLACK and $y$ and $z$ are right and left children. Invert $z$'s pa and granpa colors



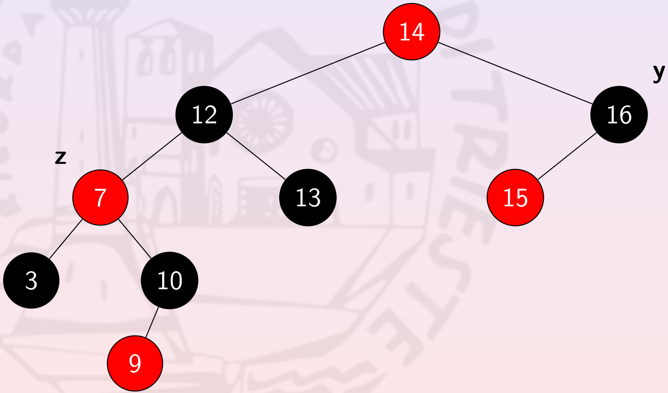
# Inserting a New Node: Example

**Case 3:** $y$ is BLACK and $y$ and $z$ are right and left children. Invert $z$'s pa and granpa colors



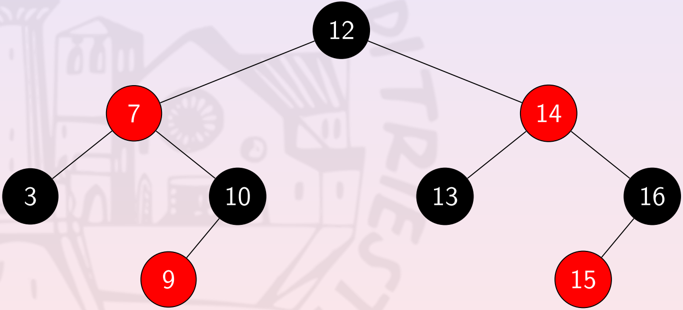
# Inserting a New Node: Example

**Case 3:** $y$ is BLACK and $y$ and $z$ are right and left children.
Invert $z$'s pa and granpa colors and rotate right on $z$'s granpa



# Inserting a New Node: Example

**Case 3:** $y$ is BLACK and $y$ and $z$ are right and left children.
Invert $z$'s pa and granpa colors and rotate right on $z$'s granpa



# Inserting a New Node: Complexity

If $z$ parent is BLACK: done

# Inserting a New Node: Complexity

If $z$ parent is BLACK: done

If $z$ has no uncle, either $z$ or $z.parent$ are the root: BLACK color it

## Inserting a New Node: Complexity

If $z$ parent is BLACK: done

If $z$ has no uncle, either $z$ or $z.parent$ are the root: BLACK color it

If $z$ has an uncle, we can always choose between Cases 1, 2, or 3

# Inserting a New Node: Complexity

If $z$ parent is BLACK: done

If $z$ has no uncle, either $z$ or $z.parent$ are the root: BLACK color it

If $z$ has an uncle, we can always choose between Cases 1, 2, or 3

Case 1 either removes the problem or pushes it towards the root

# Inserting a New Node: Complexity

If $z$ parent is BLACK: done

If $z$ has no uncle, either $z$ or $z.parent$ are the root: BLACK color it

If $z$ has an uncle, we can always choose between Cases 1, 2, or 3

Case 1 either removes the problem or pushes it towards the root

Case 2 brings to Case 3

# Inserting a New Node: Complexity

If $z$ parent is BLACK: done

If $z$ has no uncle, either $z$ or $z.parent$ are the root: BLACK color it

If $z$ has an uncle, we can always choose between Cases 1, 2, or 3

Case 1 either removes the problem or pushes it towards the root

Case 2 brings to Case 3

Case 3 solves the problem

# Inserting a New Node: Complexity

If $z$ parent is BLACK: done

If $z$ has no uncle, either $z$ or $z.parent$ are the root: BLACK color it

If $z$ has an uncle, we can always choose between Cases 1, 2, or 3

Case 1 either removes the problem or pushes it towards the root

Case 2 brings to Case 3

Case 3 solves the problem

In the worst case, the algorithm keeps repeating Case 1 steps along the insertion branch and the complexity is $O(\log n)$

# Inserting a New Node: Code

```
def INSERT_RBTREE(T, z):
  INSERT_BST(T, z)
  z.color ← RED

  FIX_INSERT_RBTREE(T, z)
enddef

def FIX_INSERT_RBT_CASE1(T, z):
  UNCLE(z).color ← BLACK
  z.parent.color ← BLACK
  GRANDPARENT(z).color ← RED

  return GRANDPARENT(z)
endif
```

# Inserting a New Node: Code (Cont'd)

```
def FIX_INSERT_RBT_CASE2(T,z):
  p ← z.parent

  z_side ← CHILDHOOD_SIDE(z)
  ROTATE(T,p,REVERSE_SIDE(z_side))

  return p
endif
```

# Inserting a New Node: Code (Cont'd 2)

```
def FIX_INSERT_RBT_CASE3(T, z):
  g ← GRANDPARENT(z)

  z.parent.color ← BLACK
  g.color ← RED

  ROTATE(T, g, CHILDHOOD_SIDE(z))
endif
```

# Inserting a New Node: Code (Cont'd 3)

```
def FIX_INSERT_RBTREE(T,z):
  while (z.parent≠NIL and
         (GRANDPARENT(z)≠NIL or z.parent.color = RED)):
    if UNCLE(z).color = RED:
      z ← FIX_INSERT_RBT_CASE1(T,z)
    else:
      if (CHILDHOOD_SIDE(z) ≠
            CHILDHOOD_SIDE(z.parent)):
        z ← FIX_INSERT_RBT_CASE2(T,z)
      endif
      FIX_INSERT_RBT_CASE3(T,z)
    endif
  endwhile

  T.root.color ← BLACK
enddef
```

# Removing a Key

Removing a key as in BST removes also a node $y$ which is replaced by its former child $x$

# Removing a Key

Removing a key as in BST removes also a node $y$ which is replaced by its former child $x$

If $y$ was RED, the RB-Tree properties are preserved

# Removing a Key

Removing a key as in BST removes also a node $y$ which is replaced by its former child $x$

If $y$ was RED, the RB-Tree properties are preserved

If $y$ was BLACK, the branches through $x$ lost 1 BLACK node

In particular, $BH(x) = BH(w) - 1$ where $w$ is $x$'s sibling

# Removing a Key

Removing a key as in BST removes also a node $y$ which is replaced by its former child $x$

If $y$ was RED, the RB-Tree properties are preserved

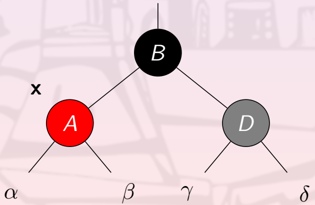
If $y$ was BLACK, the branches through $x$ lost 1 BLACK node

In particular, $BH(x) = BH(w) - 1$ where $w$ is $x$'s sibling

The fixing procedure iteratively balances BH on the sub-tree rooted on $x$'s parent

# Removing a Key: Case 0

$x$ is RED

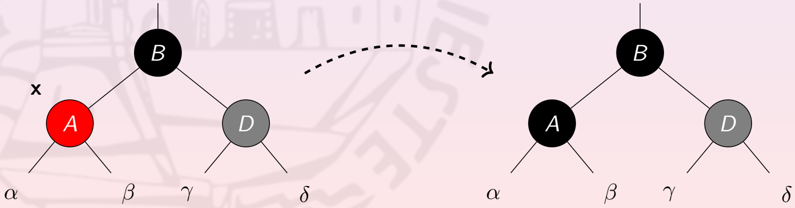


# Removing a Key: Case 0

$x$ is RED

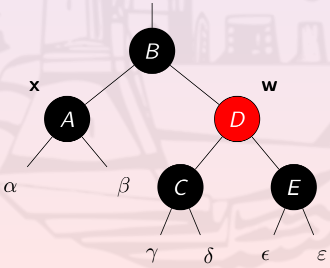
- BLACK color $x$

$BH(x)$ is increased by 1 and the tree has been fixed



# Removing a Key: Case 1

$x$'s sibling is RED

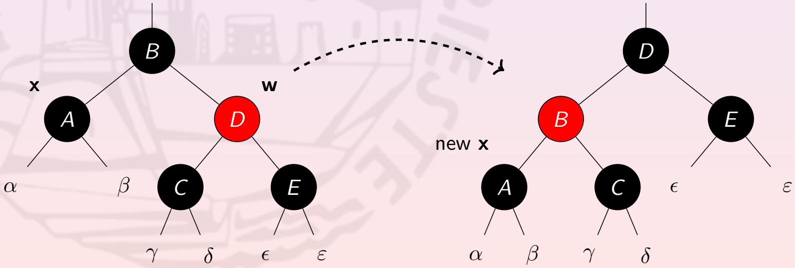


# Removing a Key: Case 1

$x$'s sibling is RED

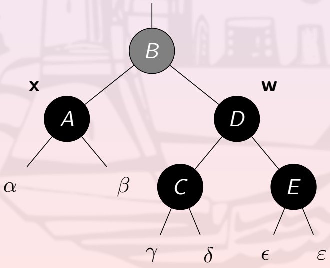
- invert colors in $x$'s parent and sibling
- rotate $x$'s parent on $x$'s side

$BH(x)$ does not change



# Removing a Key: Case 2

$x$'s sibling and nephews are BLACK

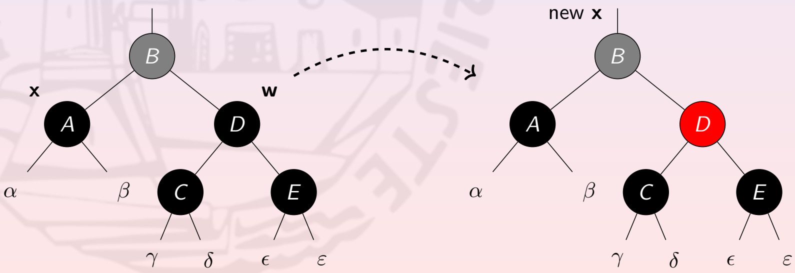


# Removing a Key: Case 2

$x$'s sibling and nephews are BLACK

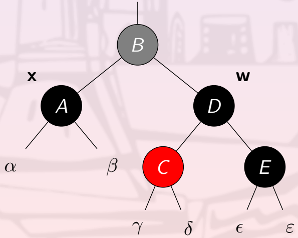
- RED color $x$'s sibling

$BH(x)$ does not change, while the BLACK height of both $x$'s parent and sibling are decreased by 1



# Removing a Key: Case 3

Among $x$'s sibling and nephews, only the nephew on $x$'s side is RED

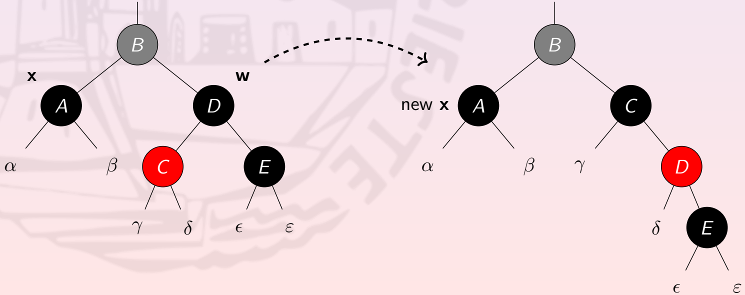


# Removing a Key: Case 3

Among $x$'s sibling and nephews, only the nephew on $x$'s side is RED

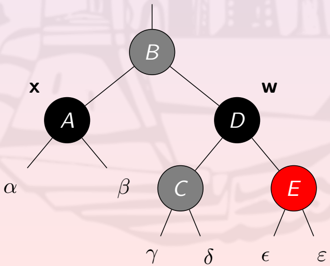
- rotate $x$'s sibling on the opposite side w.r.t. $x$
- invert colors in both old and new siblings of $x$

The BLACK height of both $x$ and $x$'s parent does not change



# Removing a Key: Case 4

The $x$'s nephew on the opposite side w.r.t. $x$ is RED

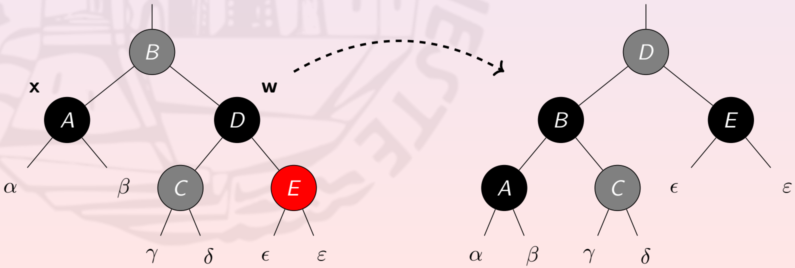


# Removing a Key: Case 4

The $x$'s nephew on the opposite side w.r.t. $x$ is RED

- switch colors of $x$'s parent and sibling
- BLACK color the $x$'s nephew on the opposite side w.r.t. $x$
- rotate $x$'s parent on $x$'s side

The BLACK height of both $x$ and $x$'s parent does not change



# Removing a Key: Some Considerations

All the case transformation procedures takes time $\Theta(1)$

# Removing a Key: Some Considerations

All the case transformation procedures takes time $\Theta(1)$

Both Case 0 and Case 4 transformation procedures fix the RB-Tree

# Removing a Key: Some Considerations

All the case transformation procedures takes time $\Theta(1)$

Both Case 0 and Case 4 transformation procedures fix the RB-Tree

Case 1 cannot occur twice in-row

# Removing a Key: Some Considerations

All the case transformation procedures takes time $\Theta(1)$

Both Case 0 and Case 4 transformation procedures fix the RB-Tree

Case 1 cannot occur twice in-row

Case 2 pushes the problem one step towards the root

# Removing a Key: Some Considerations

All the case transformation procedures takes time $\Theta(1)$

Both Case 0 and Case 4 transformation procedures fix the RB-Tree

Case 1 cannot occur twice in-row

Case 2 pushes the problem one step towards the root

If Case 2 occurs after Case 1, Case 0 occurs next

# Removing a Key: Some Considerations

All the case transformation procedures takes time $\Theta(1)$

Both Case 0 and Case 4 transformation procedures fix the RB-Tree

Case 1 cannot occur twice in-row

Case 2 pushes the problem one step towards the root

If Case 2 occurs after Case 1, Case 0 occurs next

Case 3 transformation procedure brings to Case 4

# Removing a Key: Some Considerations

In the worst case scenario, Case 2 is repeated until the problem is pushed up to the root ($O(\log n)$ times)

If this is the case, we have decreased the BLACK height of the tree and the problem is no more a problem

# Removing a Key: Code

```
def REMOVE_RBTREE(T, z):
  y ← REMOVE_BST(T, z)

  if y.color = BLACK: # fix from its replacement
    if y.left = NIL
      x ← y.right
    else:
      x ← y.left
    endif
    FIX_REMOVE_RBTREE(T, x)
  endif

  return y
enddef
```

# Removing a Key: Code (Cont'd 2)

```
def FIX_REMOVE_RBT_CASE1(T,x):
  SIBLING(x).color ← BLACK
  x.parent.color ← RED

  ROTATE(T,x,CHILDHOOD_SIDE(x))
endif

def FIX_REMOVE_RBT_CASE2(T,x):
  SIBLING(x).color ← RED
  return x.parent
endif
```

# Removing a Key: Code (Cont'd 3)

```
def FIX_REMOVE_RBT_CASE3(T,x):
  x_side ← CHILDHOOD_SIDE(x)
  r_side ← REVERSE_SIDE(x_side)

  w← GET_CHILD(w, r_side)
  GET_CHILD(w, x_side).color ← BLACK
  w.color ← RED

  ROTATE(T,w, r_side)
endif
```

# Removing a Key: Code (Cont'd 4)

```
def FIX_REMOVE_RBT_CASE4(T, x):
  x_side ← CHILDHOOD_SIDE(x)
  r_side ← REVERSE_SIDE(x_side)

  w← GET_CHILD(w, r_side)
  GET_CHILD(w, r_side).color ← BLACK
  w.color ← x.parent.color
  x.parent.color ← BLACK

  ROTATE(T, x.parent, x_side)
endif
```

# Removing a Key: Code (Cont'd 5)

```
def FIX_REMOVE_RBT(T,x):
  while x ≠ T.root and x.color ≠ RED:
    w ← SIBLING(x)
    if w = RED:
      x ← FIX_REMOVE_RBT_CASE1(T,x)
    else:
      x_side ← CHILDHOOD_SIDE(x)
      r_side ← REVERSE_SIDE(x_side)
```

# Removing a Key: Code (Cont'd 6)

```
          if GET_CHILD(w, r_size) = RED:
            FIX_REMOVE_RBT_CASE4(T,x)

            return
          else:
            if GET_CHILD(w, x_side) = RED:
              FIX_REMOVE_RBT_CASE3(T,x)
            else:
              FIX_REMOVE_RBT_CASE2(T,x)
            endif
        endif
      endwhile
    enddef
```