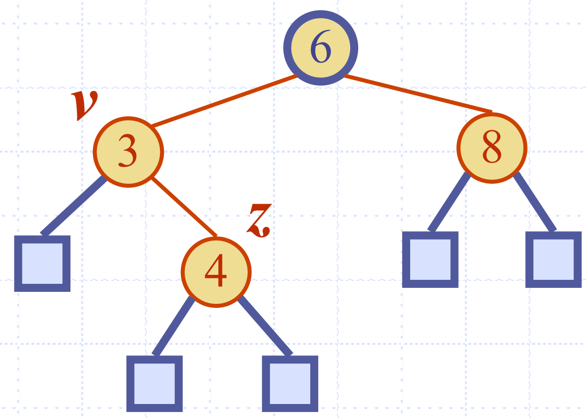
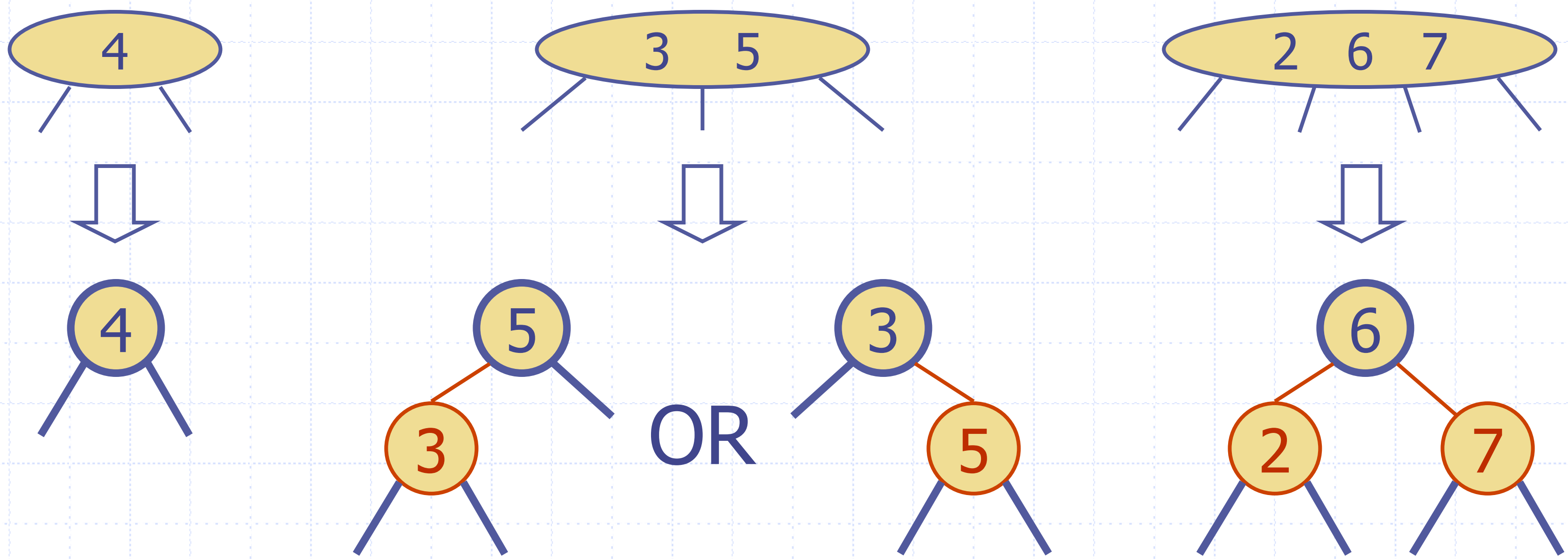
Red-Black Trees



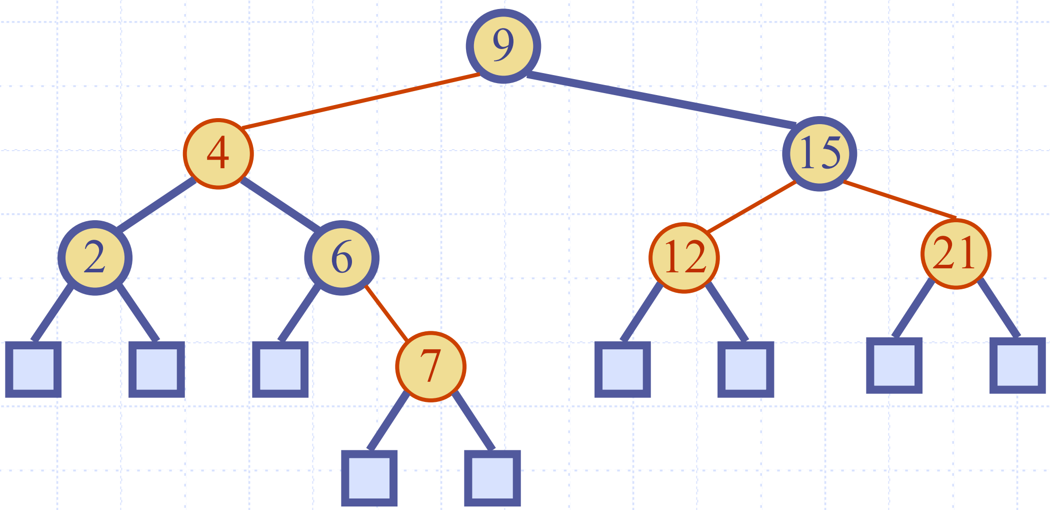
From (2,4) to Red-Black Trees

- A red-black tree is a representation of a (2,4) tree by means of a binary tree whose nodes are colored red or **black**
- In comparison with its associated (2,4) tree, a red-black tree has
  - same logarithmic time performance
  - simpler implementation with a single node type



Red-Black Trees

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth



class RedBlackTreeMap

```
class RedBlackTreeMap( TreeMap ):

class _Node( TreeMap._Node ):
    __slots__ = '_red'

    def __init__( self, element, parent = None, left = None, right = None ):
        super().__init__( element, parent, left, right )
        self._red = True

def _set_red( self, p ): p._node._red = True
def _set_black( self, p ): p._node._red = False
def _set_color( self, p, make_red ): p._node._red = make_red
def _is_red( self, p ): return p is not None and p._node._red
def _is_red_leaf( self, p ): return self._is_red( p ) and self.is_leaf( p )
```

Height of a Red-Black Tree

- Theorem: A red-black tree storing n items has height $O(\log n)$

  Proof:
  - The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$
- The search algorithm for a binary search tree is the same as that for a binary search tree
- By the above theorem, searching in a red-black tree takes $O(\log n)$ time

Insertion (case root)

Root Property: the root is black

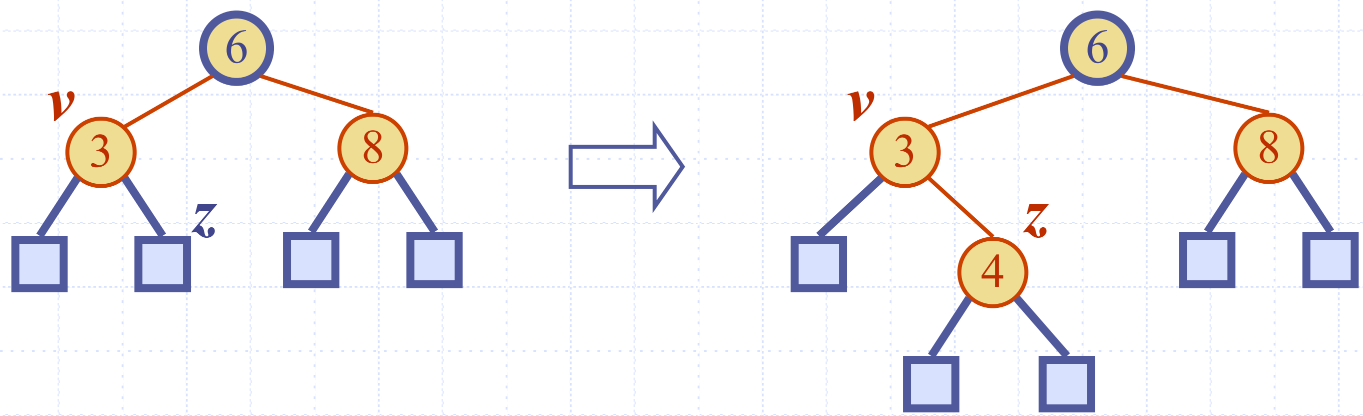
```
def _rebalance_insert( self, p ):
    #new node always red
    self._resolve_red( p )

def _resolve_red( self, p ):
    if self.is_root( p ):
        #make root black
        self._set_black( p )
    else:
        ...
```

Insertion

Internal Property: the children of a red node are black

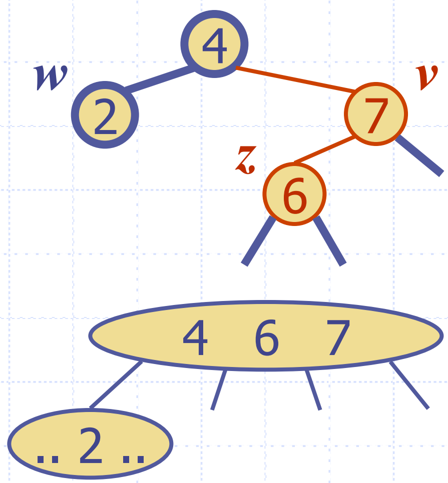
- To insert (k, o), we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root
  - We preserve the root, external, and depth properties
  - If the parent v of z is black, we also preserve the internal property and we are done
  - Else (v is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
- Example where the insertion of 4 causes a double red:



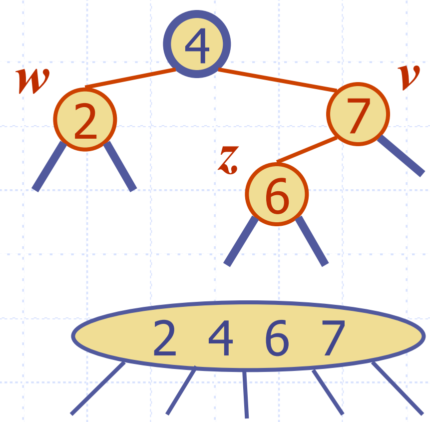
Remedying a Double Red

- Consider a double red with child *z* and parent *v*, and let *w* be the sibling of *v*

Case 1: *w* is black
- The double red is an incorrect replacement of a 4-node
- Restructuring: we change the 4-node replacement

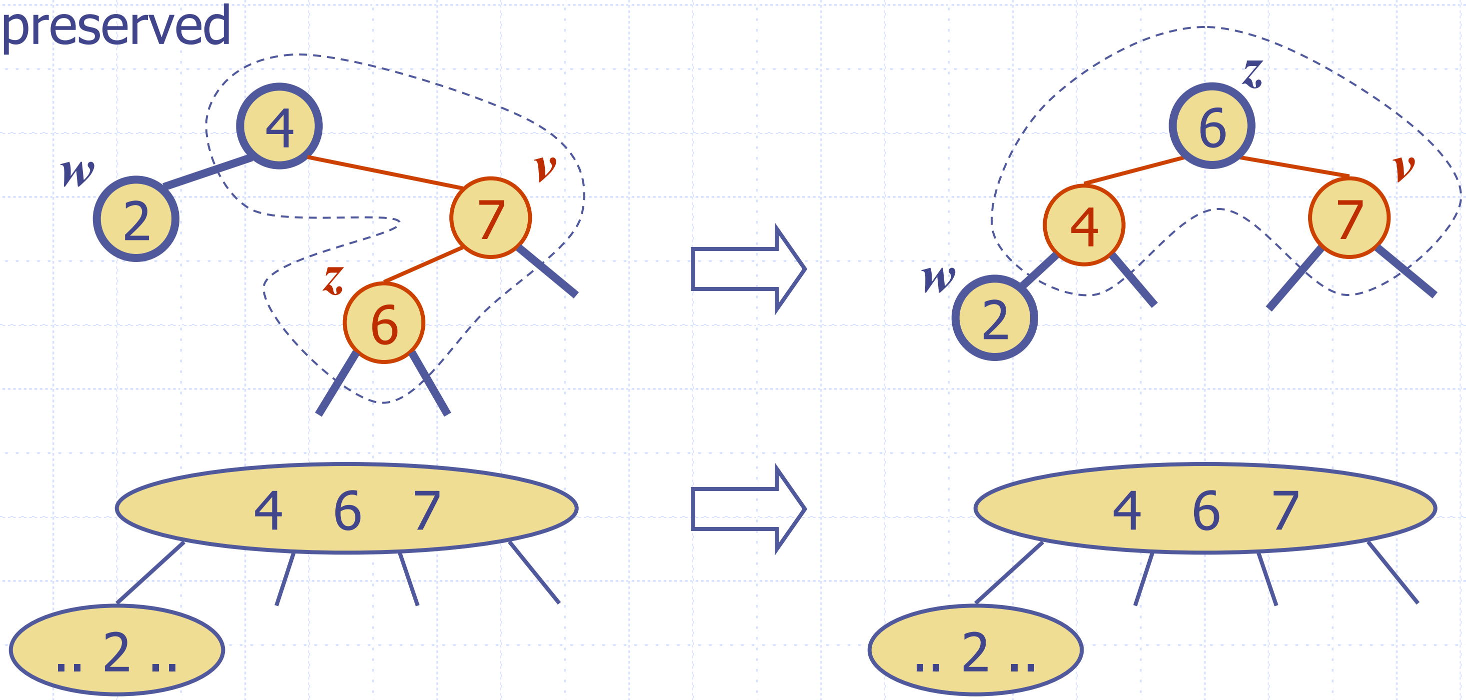


Case 2: *w* is red
- The double red corresponds to an overflow
- Recoloring: we perform the equivalent of a split



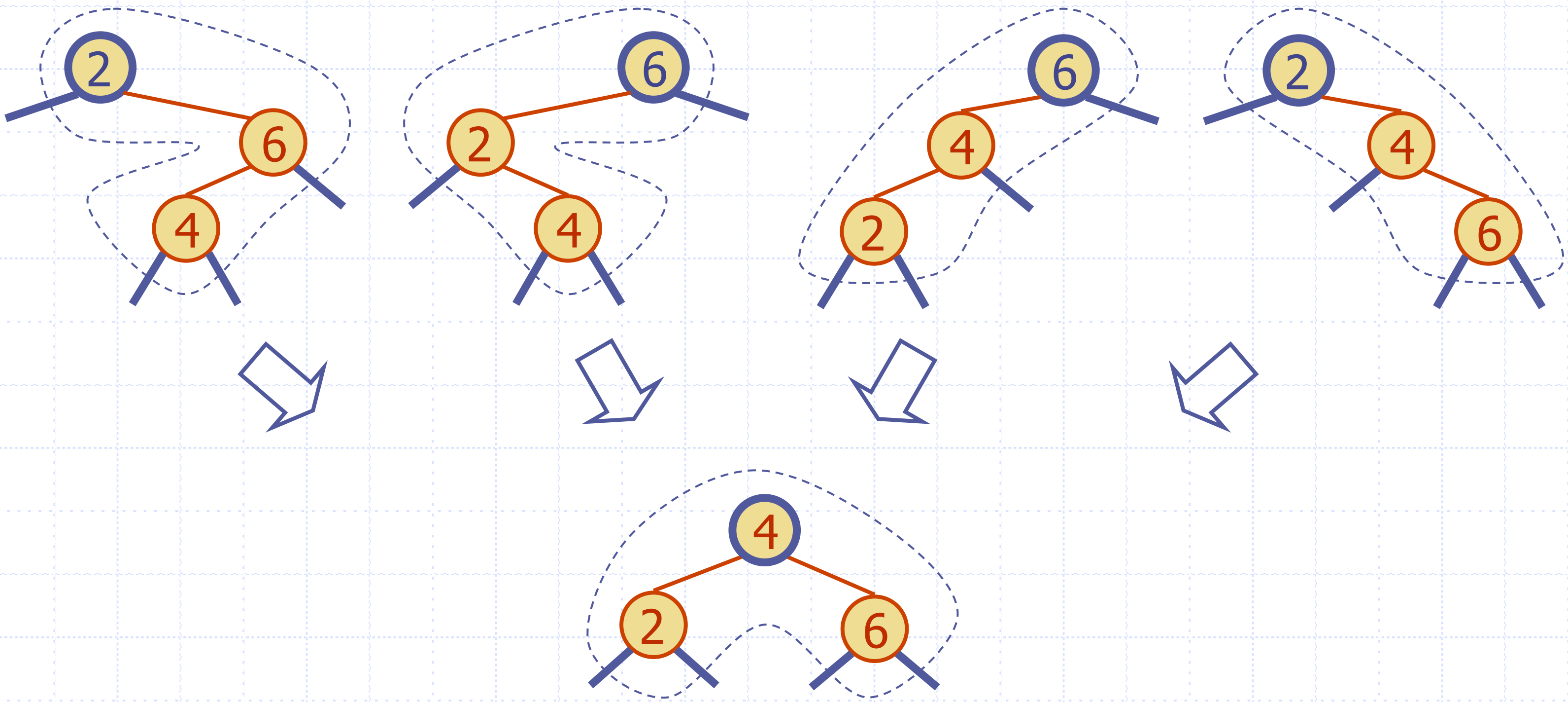
Restructuring

- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- It is equivalent to restoring the correct replacement of a 4-node
- The internal property is restored and the other properties are preserved



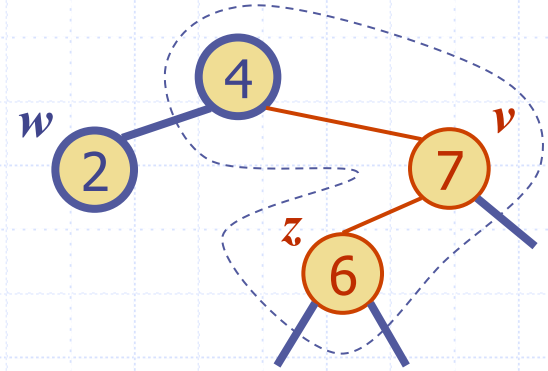
Restructuring (cont.)

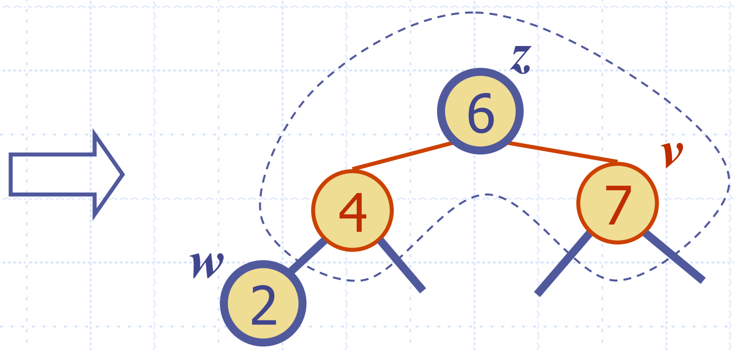
- There are four restructuring configurations depending on whether the double red nodes are left or right children



Insertion (case 1: black uncle)

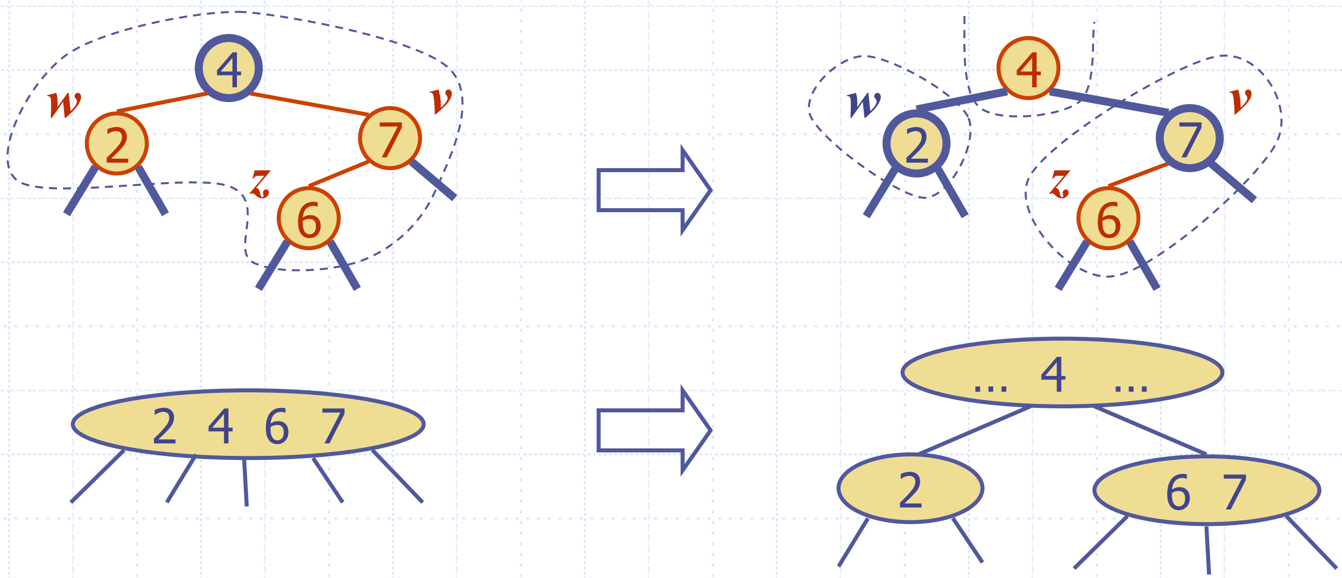
```
else:
    parent = self.parent( p )
    if self._is_red( parent ):
        #double red problem
        uncle = self.sibling( parent )
        if not self._is_red( uncle ):
            #Case 1: misshapen 4-node
            middle = self._restructure( p ) #do trinode restructuring
            self._set_black( middle )       #and then fix colors
            self._set_red( self.left( middle ) )
            self._set_red( self.right( middle ) )
            #uncle was already black from the case
            ...
```



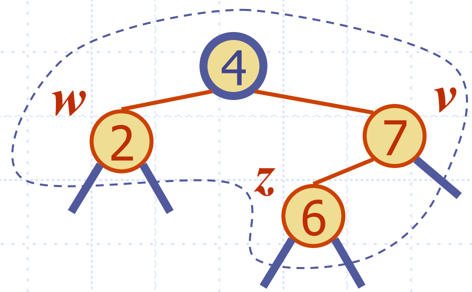


Recoloring

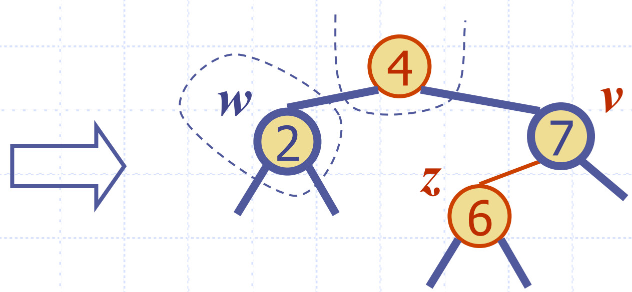
- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate to the grandparent u



Insertion (case 2: red uncle)




```
else:
    #Case 2: overflow (5-node)
    grand = self.parent( parent )
    self._set_red( grand )                   #grandparent becomes red
    self._set_black( self.left( grand ) )    #its children become black
    self._set_black( self.right( grand ) )
    self._resolve_red( grand )               #recur at red grandparent
```



Analysis of Insertion

Algorithm *insert*(k, o)

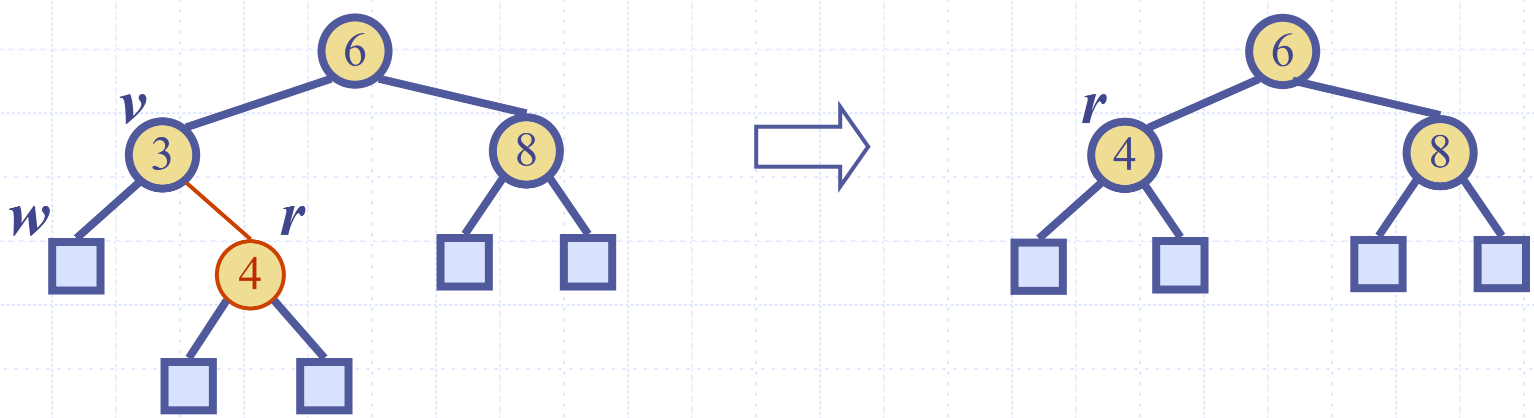
1. We search for key k to locate the insertion node z
2. We add the new entry (k, o) at node z and color z red

3. **while** *doubleRed*(z)
   if *isBlack*(*sibling*(*parent*(z)))
      z = *restructure*(z)
      return
   else { *sibling*(*parent*(z) is red }
      z = *recolor*(z)

- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because we perform
  - $O(\log n)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$ time
- Thus, an insertion in a red-black tree takes $O(\log n)$ time

Deletion (the easy cases)

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees (results in the removal of a node that has at most one child, (either the node containing k or its inorder predecessor) and the promotion of its remaining child (if any).
- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - If either v or r was red, we color r black and we are done (corresponds to shrinking a 3-node or a 4-node).
  - When v black, then it either has zero children or it has one red leaf child (because the null subtree of the removed node has black height 0).
- Examples deletion of 3:

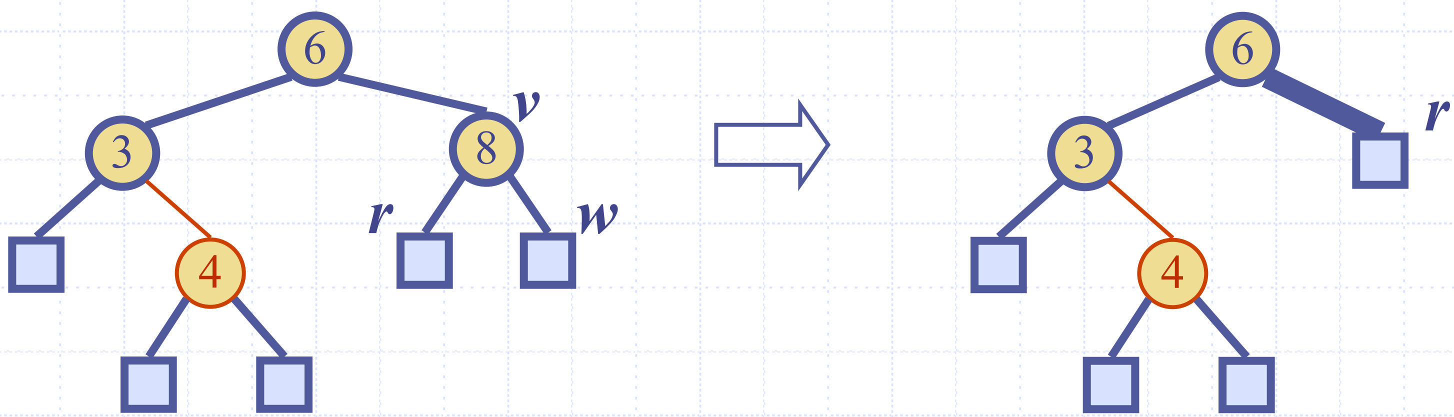


Deletion code

```
def _rebalance_delete( self, p ):
    #p is the parent of the node to be deleted (from TreeMap.py)
    if len( self ) == 1:
        #special case: ensure root is black
        self._set_black( self._root() )
    elif p is not None:
        n = self.num_children( p )
        if n == 1:
            #not a problem if red leaf, otherwise double black
            c = next( self.children( p ) )
            if not self._is_red_leaf( c ):
                self._fix_deficit( p, c )
        elif n == 2:
            #color black the promoted red leaf
            if self._is_red_leaf( self.left( p ) ):
               self._set_black( self.left( p ) )
            else:
                self._set_black( self.right( p ) )
```

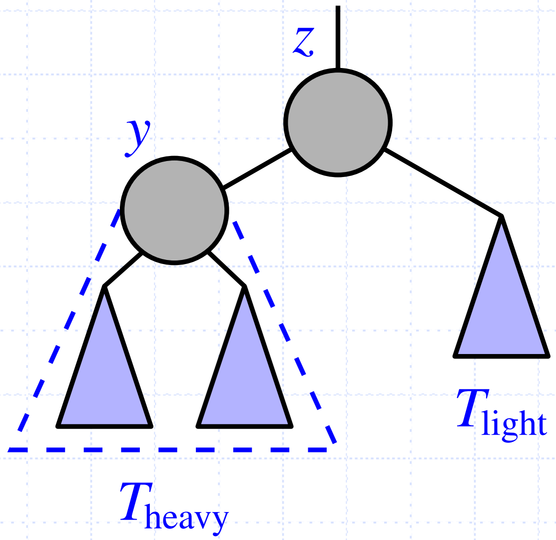
Deletion (the double black case)

- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - Else (v and r were both black) we color r ***double black***, which is a violation of the internal property requiring a reorganization of the tree WHY?
- Example where the deletion of 8 causes a double black:



Remedying a Double Black

- The algorithm for remedying a double black node w with sibling y

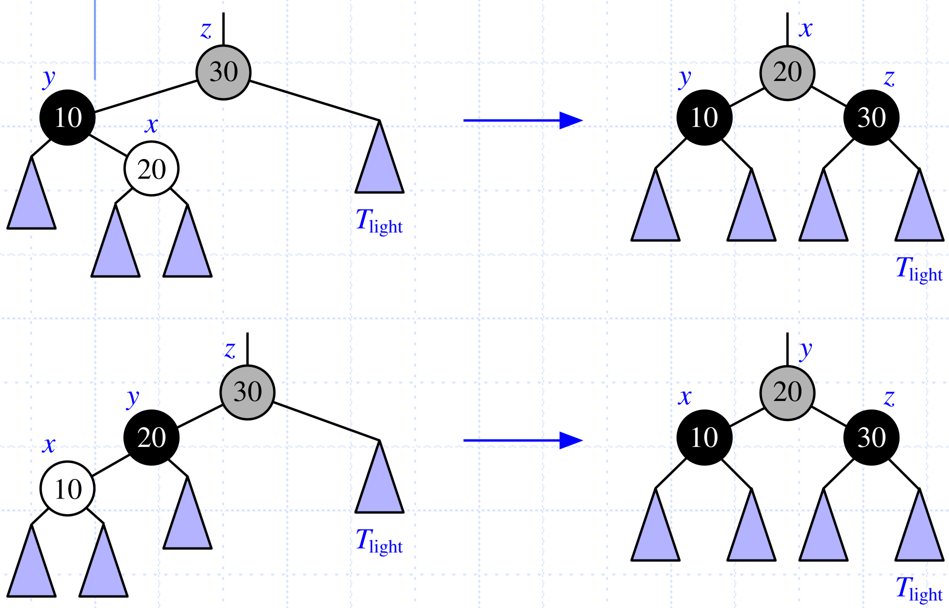


Remedying a Double Black

- The algorithm for remedying a double black node w with sibling y considers three cases

Case 1: y is black and has a red child

- We perform a restructuring, equivalent to a transfer, and we are done



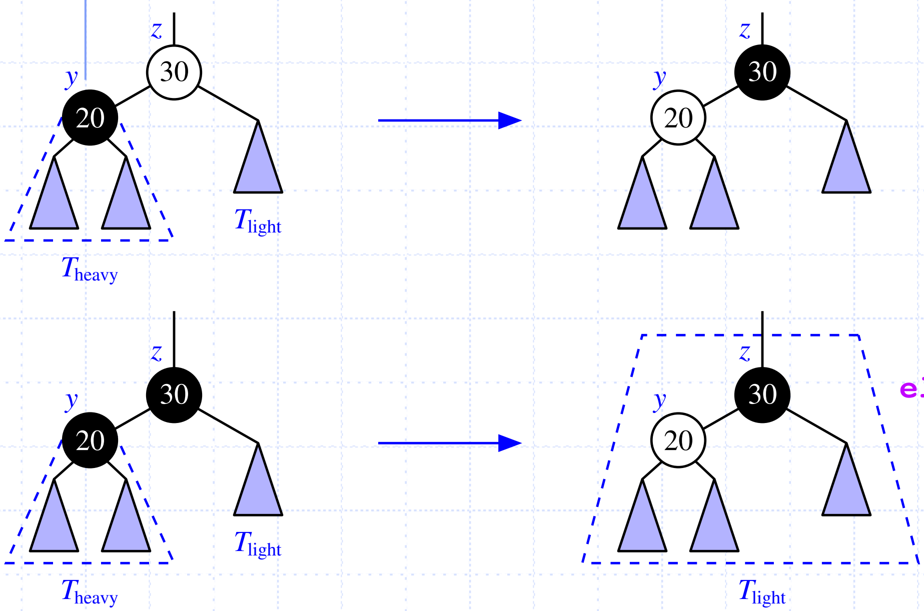

```
def _fix_deficit( self, z, y ):
    #resolve black deficit at z, where y is the
    #root of z's heavier subtree
    if not self._is_red( y ):
        #y is black; will apply Case 1 or 2
        x = self._get_red_child( y )
        if x is not None:
            #Case 1: y is black and has
            #red child x; do "transfer"
            old_color = self._is_red( z )
            middle = self._restructure( x )
            self._set_color( middle, old_color )
            self._set_black( self.left( middle ) )
            self._set_black( self.right( middle ) )
        else:
            ...
```

Remedying a Double Black

- The algorithm for remedying a double black node w with sibling y considers three cases

Case 2: y is black and its children are both black

- We perform a recoloring, equivalent to a fusion, which may propagate up the double black violation



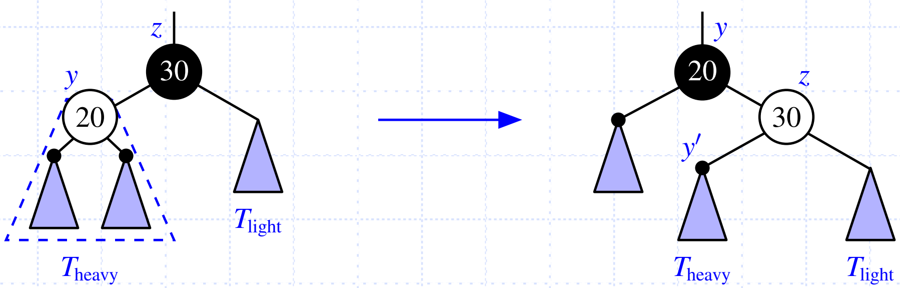

```
else:
    #Case 2: y is black, but no red children;
    #recolor as "fusion"
    self._set_red( y )
    if self._is_red( z ):
        self._set_black( z )
        #this resolves the problem
    elif not self.is_root( z ):
        #recur upward
        self._fix_deficit( self.parent( z ),
                           self.sibling( z ) )
else:
    ...
```

Remedying a Double Black

- The algorithm for remedying a double black node w with sibling y considers three cases

  Case 3: y is red

  - We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies




```
#Case 3: y is red; rotate misaligned
#3-node and repeat
self._rotate( y )
self._set_black( y )
self._set_red( z )
if z == self.right( y ):
    self._fix_deficit( z, self.left( z ) )
else:
    self._fix_deficit( z, self.right( z ) )
```

Remedying a Double Black

- The algorithm for remedying a double black node w with sibling y considers three cases
- Deletion in a red-black tree takes $O(\log n)$ time

Red-Black Tree Reorganization

Insertion	remedy double red	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	change of 4-node representation	double red removed
recoloring	split	double red removed or propagated up

Deletion	remedy double black	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	transfer	double black removed
recoloring	fusion	double black removed or propagated up
adjustment	change of 3-node representation	restructuring or recoloring follows

Python Implementation

```
class RedBlackTreeMap(TreeMap):
  """Sorted map implementation using a red-black tree."""
  class _Node(TreeMap._Node):
    """Node class for red-black tree maintains bit that denotes color."""
    __slots__ = '_red'     # add additional data member to the Node class

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._red = True     # new node red by default
```

Python Implementation, Part 2

```
#------------------------ positional-based utility methods ------------------------
# we consider a nonexistent child to be trivially black
def _set_red(self, p): p._node._red = True
def _set_black(self, p): p._node._red = False
def _set_color(self, p, make_red): p._node._red = make_red
def _is_red(self, p): return p is not None and p._node._red
def _is_red_leaf(self, p): return self._is_red(p) and self.is_leaf(p)

def _get_red_child(self, p):
  """Return a red child of p (or None if no such child)."""
  for child in (self.left(p), self.right(p)):
    if self._is_red(child):
      return child
  return None

#------------------------ support for insertions ------------------------
def _rebalance_insert(self, p):
  self._resolve_red(p)                       # new node is always red

def _resolve_red(self, p):
  if self.is_root(p):
    self._set_black(p)                       # make root black
  else:
    parent = self.parent(p)
    if self._is_red(parent):                 # double red problem
      uncle = self.sibling(parent)
      if not self._is_red(uncle):            # Case 1: misshapen 4-node
        middle = self._restructure(p)        # do trinode restructuring
        self._set_black(middle)              # and then fix colors
        self._set_red(self.left(middle))
        self._set_red(self.right(middle))
      else:                                  # Case 2: overfull 5-node
        grand = self.parent(parent)
        self._set_red(grand)                 # grandparent becomes red
        self._set_black(self.left(grand))    # its children become black
        self._set_black(self.right(grand))
        self._resolve_red(grand)             # recur at red grandparent
```

Python Implementation, end

```
#------------------------- support for deletions -------------------------
def _rebalance_delete(self, p):
  if len(self) == 1:
    self._set_black(self.root())        # special case: ensure that root is black
  elif p is not None:
    n = self.num_children(p)
    if n == 1:                          # deficit exists unless child is a red leaf
      c = next(self.children(p))
      if not self._is_red_leaf(c):
        self._fix_deficit(p, c)
    elif n == 2:                        # removed black node with red child
      if self._is_red_leaf(self.left(p)):
        self._set_black(self.left(p))
      else:
        self._set_black(self.right(p))

def _fix_deficit(self, z, y):
  """Resolve black deficit at z, where y is the root of z's heavier subtree."""
  if not self._is_red(y): # y is black; will apply Case 1 or 2
    x = self._get_red_child(y)
    if x is not None: # Case 1: y is black and has red child x; do "transfer"
      old_color = self._is_red(z)
      middle = self._restructure(x)
      self._set_color(middle, old_color)        # middle gets old color of z
      self._set_black(self.left(middle))        # children become black
      self._set_black(self.right(middle))
    else: # Case 2: y is black, but no red children; recolor as "fusion"
      self._set_red(y)
      if self._is_red(z):
        self._set_black(z)                      # this resolves the problem
      elif not self.is_root(z):
        self._fix_deficit(self.parent(z), self.sibling(z)) # recur upward
  else: # Case 3: y is red; rotate misaligned 3-node and repeat
    self._rotate(y)
    self._set_black(y)
    self._set_red(z)
    if z == self.right(y):
      self._fix_deficit(z, self.left(z))
    else:
      self._fix_deficit(z, self.right(z))
```