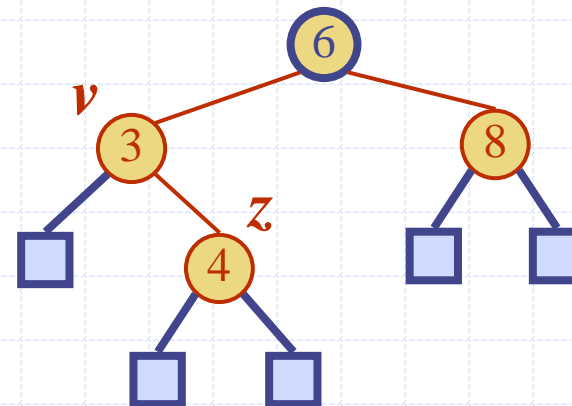


# Lecture 14: Red-Black Trees

Perfect Balance  
and Efficiency



# Wholeness Statement

A red-black tree is an implementation of a (2, 4) tree that is optimized for space utilization. The insert and delete operations are also optimized to avoid backtracking; the operations are performed locally yet maintain balance and order in the whole.

*Science of Consciousness:* Nature operates in accord with the law of least action while maintaining balance and order in the whole. Contact with the source of thought brings that order and balance out into our daily life.

# Outline and Reading

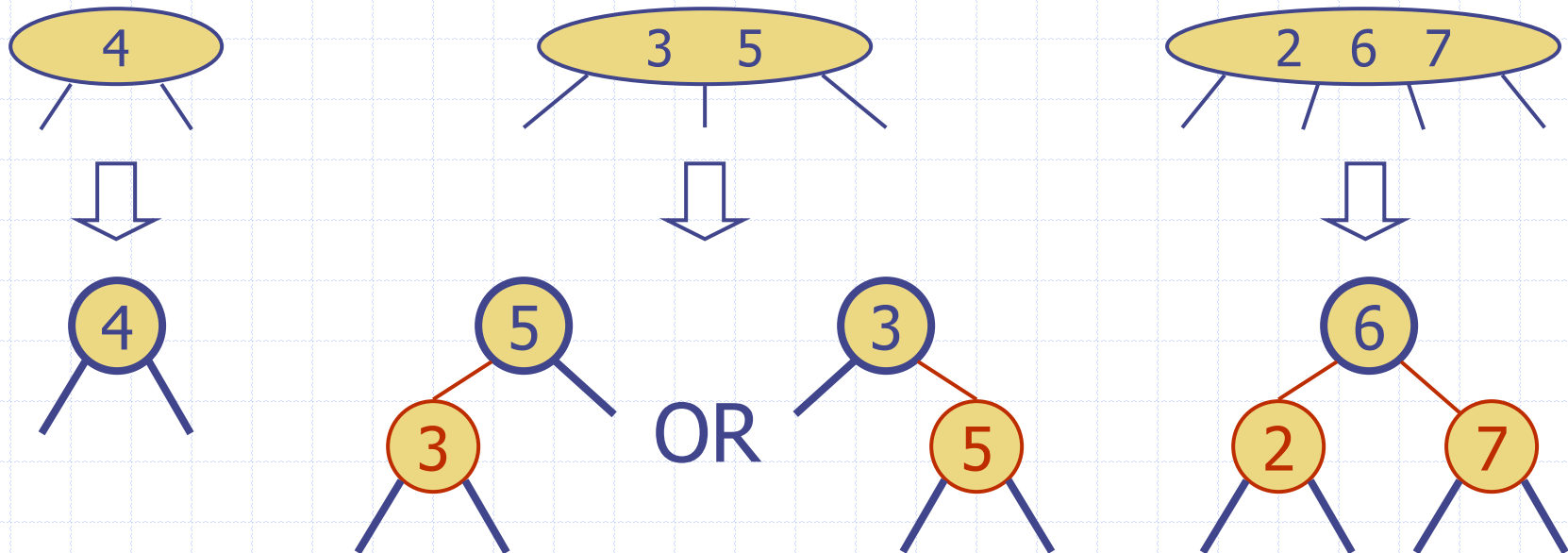
- ◆ From (2,4) trees to red-black trees
- ◆ Red-black tree
  - Definition
  - Height
  - Insertion
    - ◆ restructuring
    - ◆ recoloring
  - Deletion
    - ◆ restructuring
    - ◆ recoloring
    - ◆ adjustment

# Balanced Search Trees

- ◆ History and development of balanced search trees
  - Started with AVL trees, 1962
    - ◆ Two Soviet mathematicians (Adel'son-Vel'skiĭ and Landis)
    - ◆ Could have as many as  $O(\log n)$  rotations
  - 2-3 trees, 1970
  - B-trees, 1972
    - ◆ Generalization of 2-3 trees to any number keys per node
    - ◆ Variations (e.g., B\*-tree, B+-tree) became popular file structures
  - Symmetric binary B-trees, 1972
  - Red-Black coloring introduced, 1978
    - ◆ Became popular implementation of balanced binary trees in the late 1980's and early 1990's

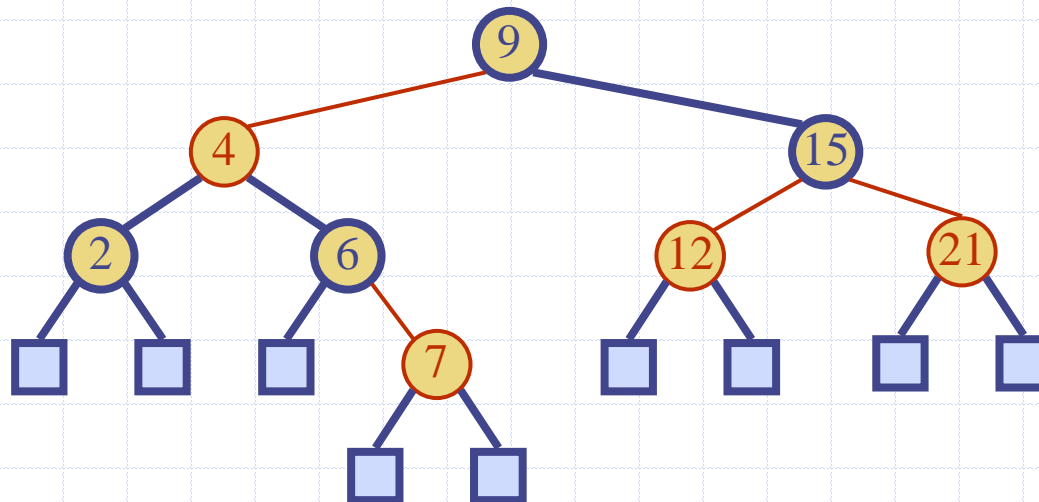
# From (2,4) to Red-Black Trees

- ◆ 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

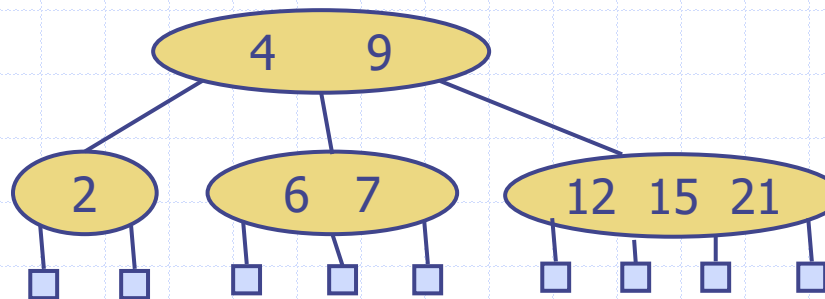
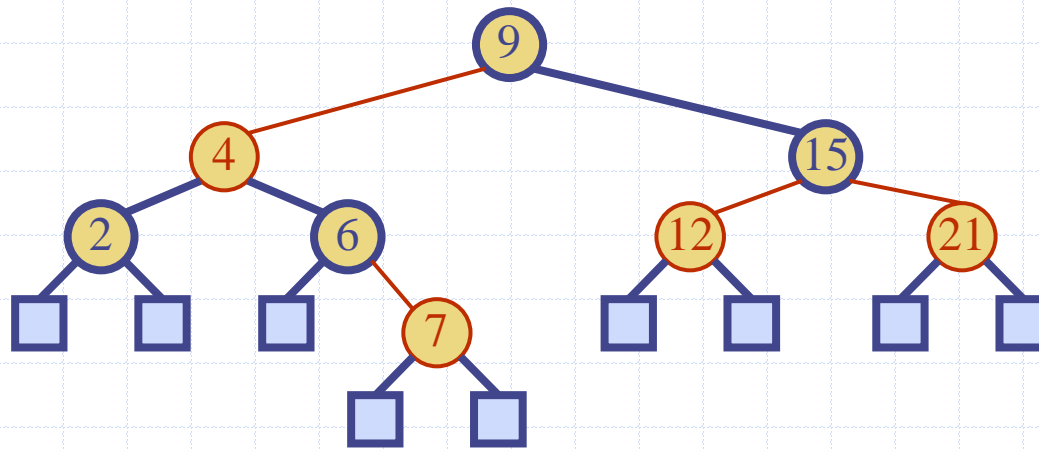


# Red-Black Tree

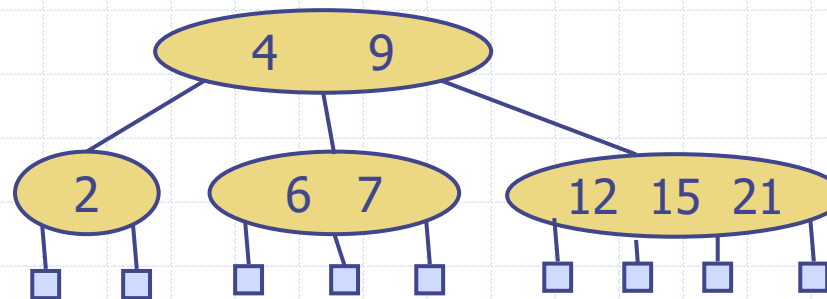
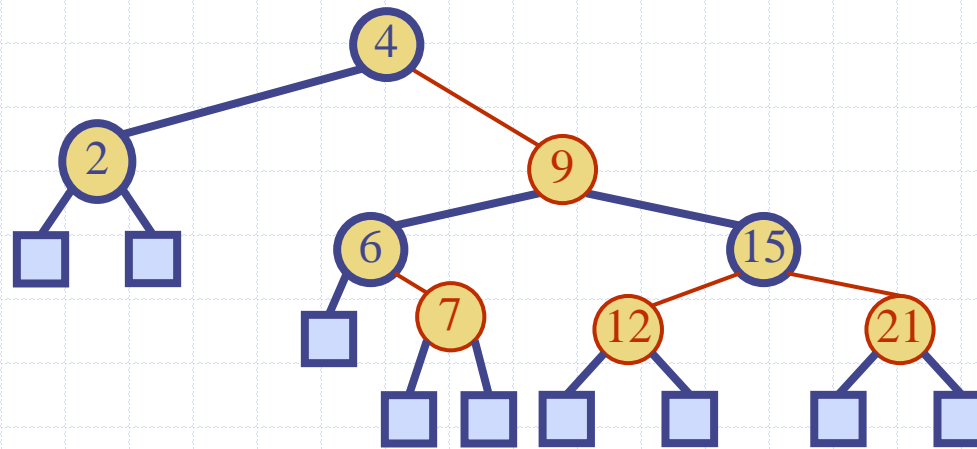
- ◆ A red-black tree can also be defined as a binary search tree that satisfies the following properties:
- **Root Property:** the root is black
  - **External Property:** every leaf is black
  - **Internal Property:** the children of a red node are black
  - **Depth Property:** all the leaves have the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



# Red-Black Tree to (2,4) Tree



# Different Red-Black Tree to Same (2,4) Tree





# 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 red-black 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

# Main Point

1. A red-black tree is an implementation of a  $(2, 4)$  tree in which each  $(2, 4)$  node is converted into a small binary tree. That is, a red-black tree is a reflection of a corresponding  $(2, 4)$  tree.

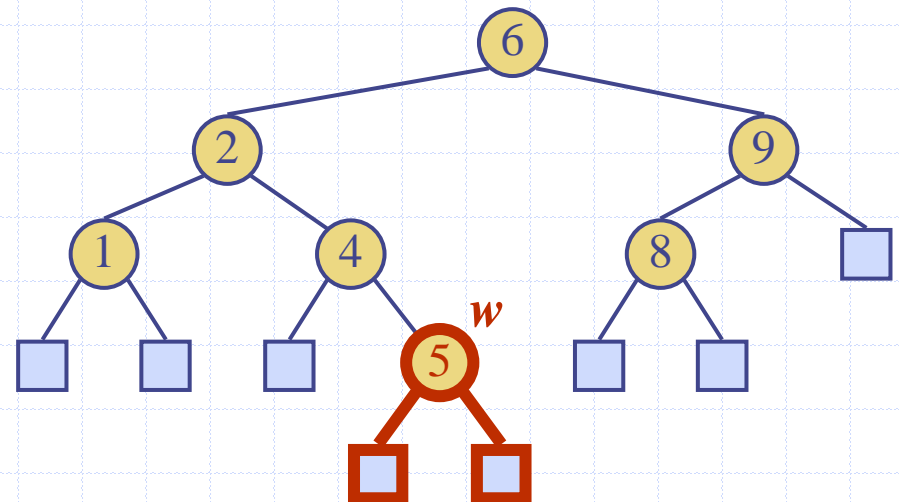
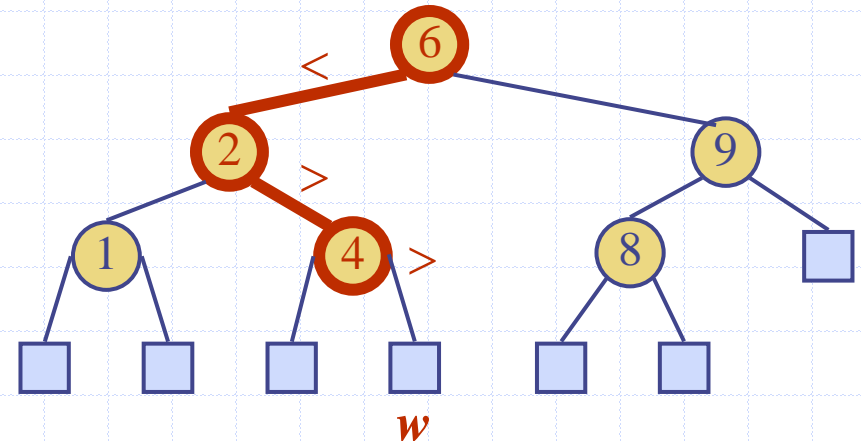
*Science of Consciousness:* Our individual consciousness is the reflection of pure consciousness based on the purity and activity of our physiology and nervous system.

# Red-Black Tree Insertion

# Recall

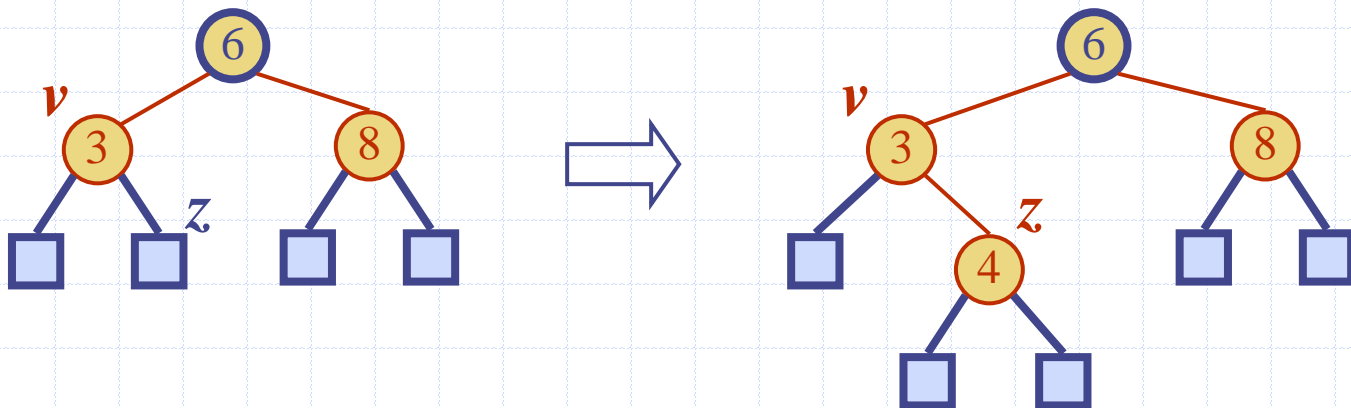
## Binary Tree Insertion

- ◆ If  $k$  is in the tree, replace the element associated with  $k$ , and return the previous element associated with  $k$
- ◆ If  $k$  is not in the tree, then a search for  $k$  will find a leaf where the item is to be inserted by operation  $\text{insert}(k, o)$
- ◆ Let  $w$  be the leaf reached by the search
- ◆ Insert  $k$  at node  $w$  and expand  $w$  into an internal node
- ◆ Example: insert 5



# Insertion

- ◆ To perform operation **insert**( $k, o$ ), we execute the insertion algorithm for binary search trees and color the newly inserted node  $z$  **red** unless it is the root. We preserve the root, external, and depth properties.
  - If the parent  $v$  of  $z$  is black, we 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
- ◆ For example, the insertion of 4 causes a double red:

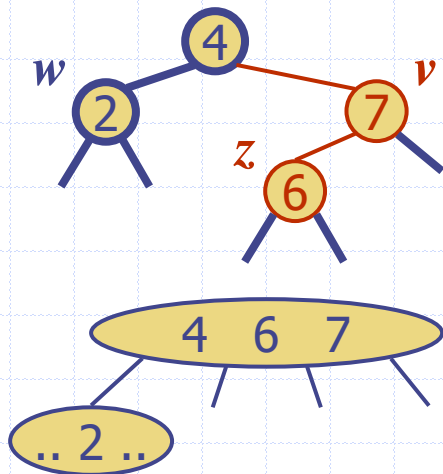


# Remedying a Double Red

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

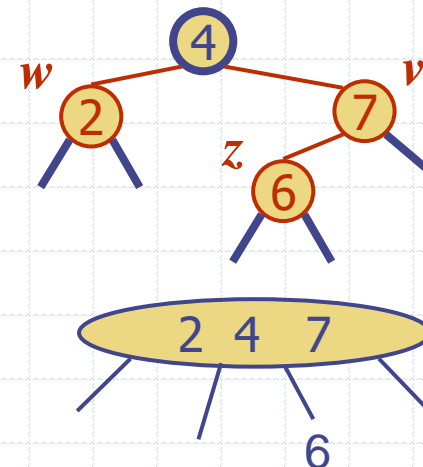
## Case 1: $z$ 's uncle $w$ is black

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



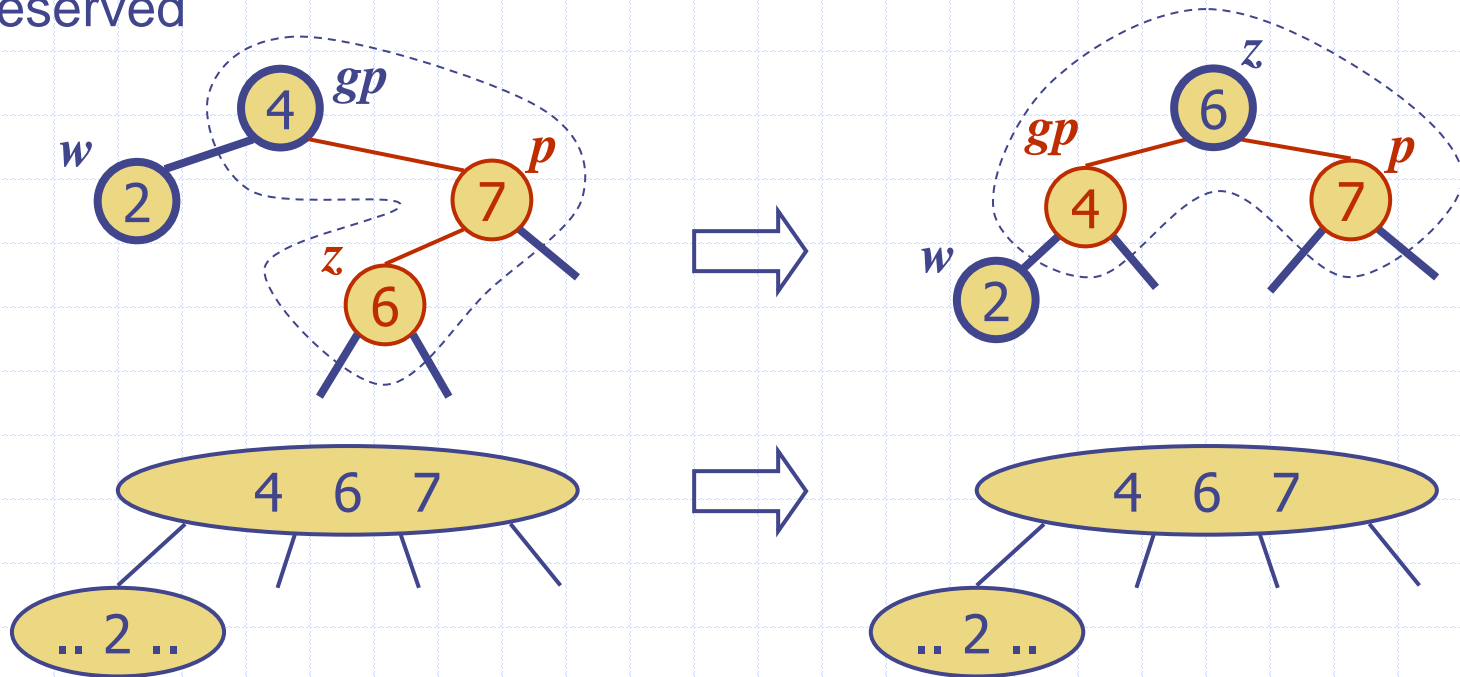
## Case 2: $z$ 's uncle $w$ is red

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



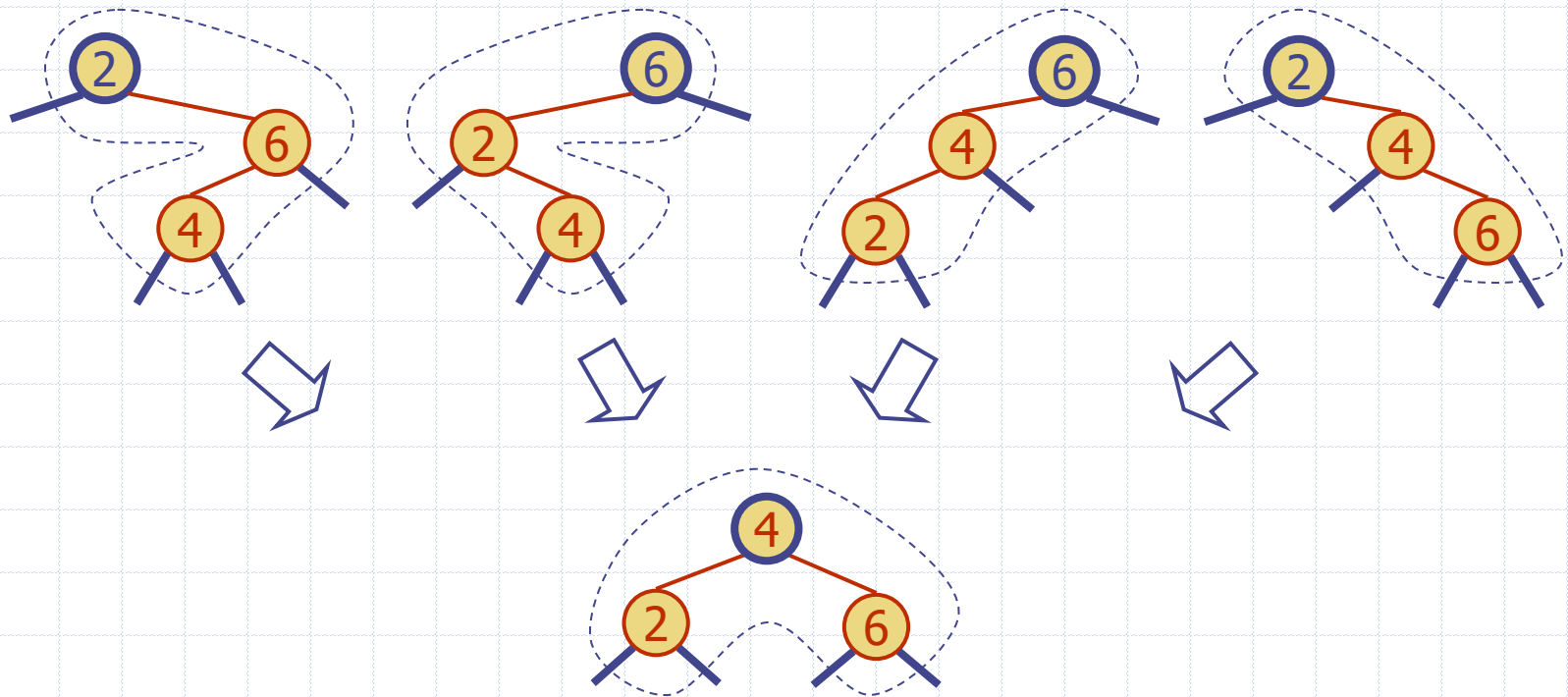
# Case 1: Restructuring

- ◆ 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.)

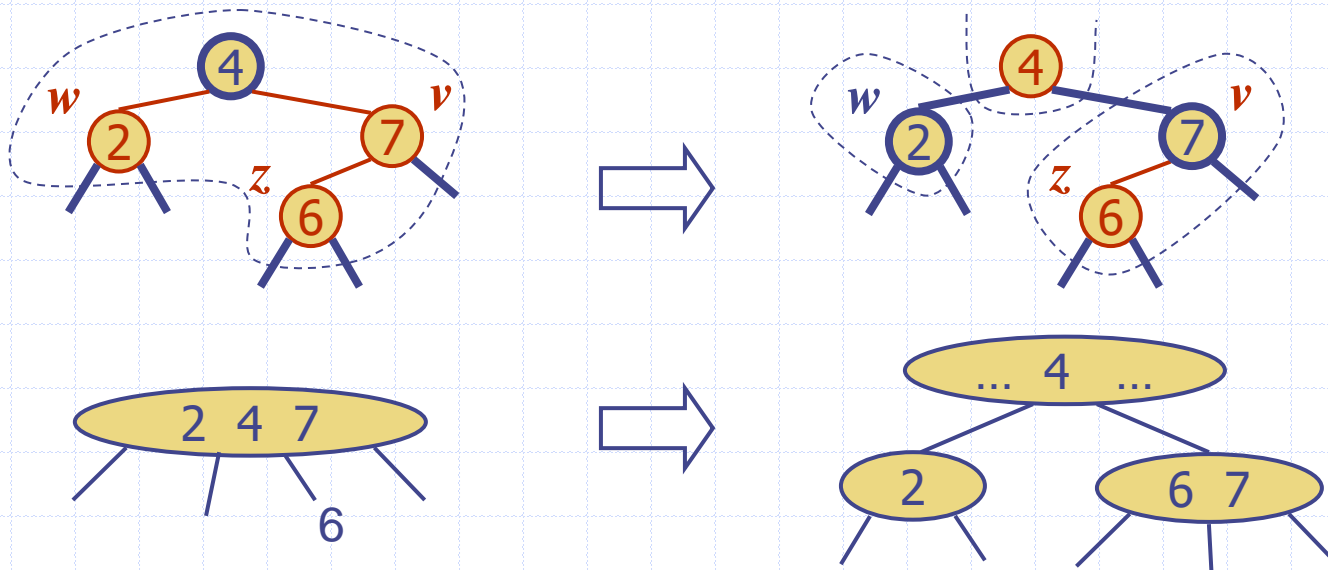
- ◆ There are four restructuring configurations depending on whether the double red nodes are left or right children





# Case 2: 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  $u$



# Example:

- ◆ Insert the following into an initially empty red-black tree in this order:  
(16, 5, 22, 45, 2, 10, 18, 30, 50, 12, 1)

# Analysis of Insertion (abstract)

## Algorithm *insertItem(k, o)*

1. Search for key  $k$  to locate the insertion node  $z$
2. Add the new item  $(k, o)$  at node  $z$  and color  $z$  red
3. **while** *isDoubleRed*( $z$ ) **do**  
    **if** *isBlack*(*sibling*(*parent*( $z$ )))  
        *restructure*( $z$ )  
        **return**  
    **else** { *sibling*(*parent*( $z$ )) is red }  
         $z \leftarrow \textit{splitRecolor}(z)$

- ◆ Recall that a red-black tree has  $O(\log n)$  height
- ◆ Step 1 takes  $O(?)$  time
- ◆ Step 2 takes  $O(?)$  time
- ◆ Step 3 takes  $O(?)$  time
- ◆ Thus, an insertion in a red-black tree takes  $O(?)$  time

# Analysis of Insertion

## Algorithm *insertItem(k, o)*

1. Search for key  $k$  to locate the insertion node  $z$  using *findPosition*
2. Add the new item  $(k, o)$  at node  $z$  and color  $z$  red
3. **while** *doubleRed(z)*  
    **if** *isBlack(sibling(parent(z)))*  
         $gpz \leftarrow \text{parent}(\text{parent}(z))$   
        *restructure(z)*  
        *setColor(gpz, RED)*  
        *setColor(parent(gpz), BLK)*  
        **return**  
    **else** { *sibling(parent(z))* is red }  
         $z \leftarrow \text{splitRecolor}(z)$

- ◆ 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

# Analysis of Insertion

Algorithm *isDoubleRed*(z)

```
if isRoot(z) then
    setColor(z, BLK)
    return False
else
    return isRed(parent(z))
```

Algorithm *splitRecolor*(z)

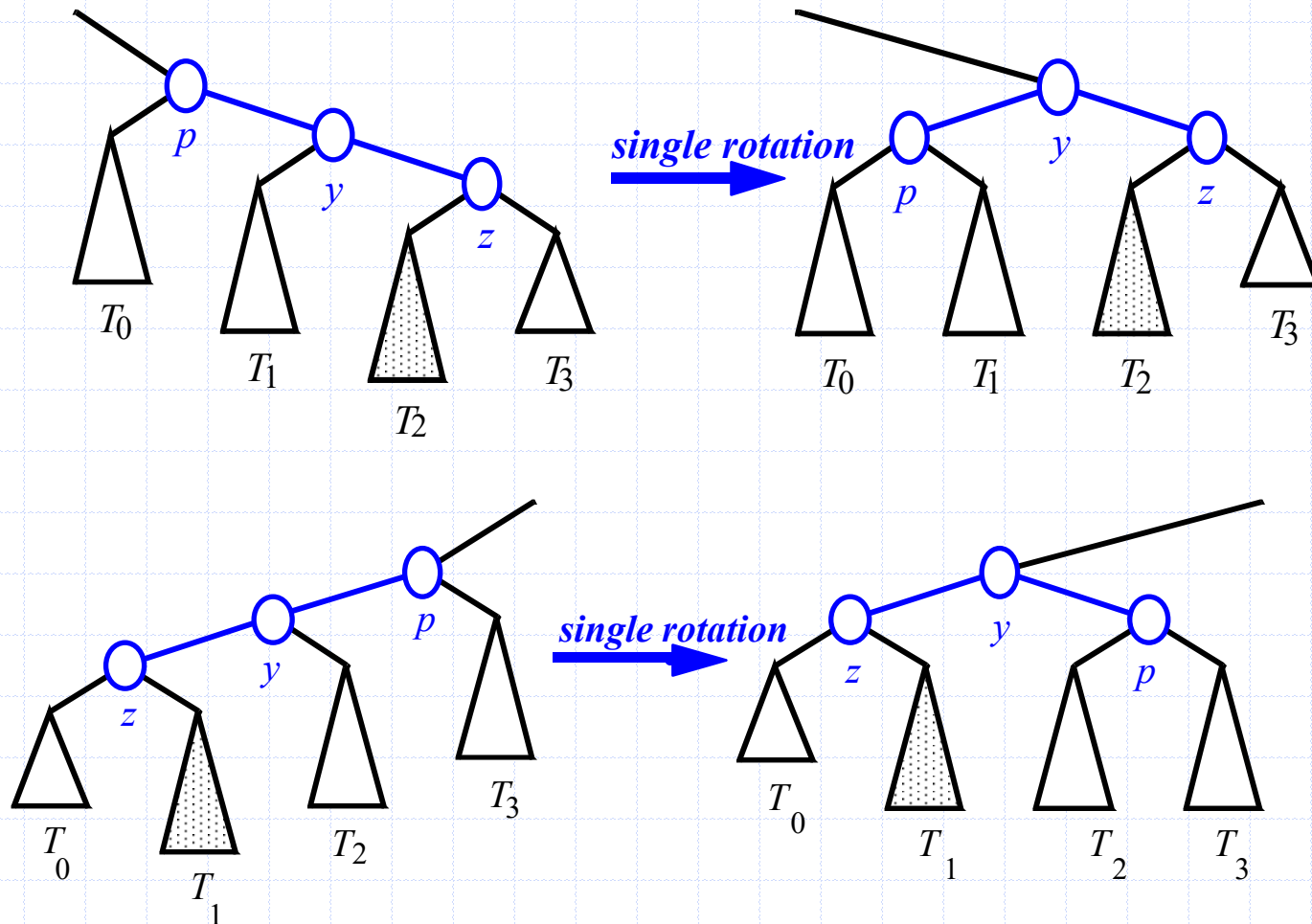
```
pz ← parent(z)
setColor(pz, BLK)
setColor(sibling(pz), BLK)
gpz ← parent(pz)
setColor(gpz, RED)
return gpz
```

Algorithm *restructure*(z)

```
pz ← parent(z)
if isLeft(z) then
    if isLeft(pz) then
        rotateRight(pz)
    else
        rotateRight(z)
        rotateLeft(z)
else { z is a right child }
    if isLeft(pz) then
        rotateLeft(z)
        rotateRight(z)
    else
        rotateLeft(pz)
```

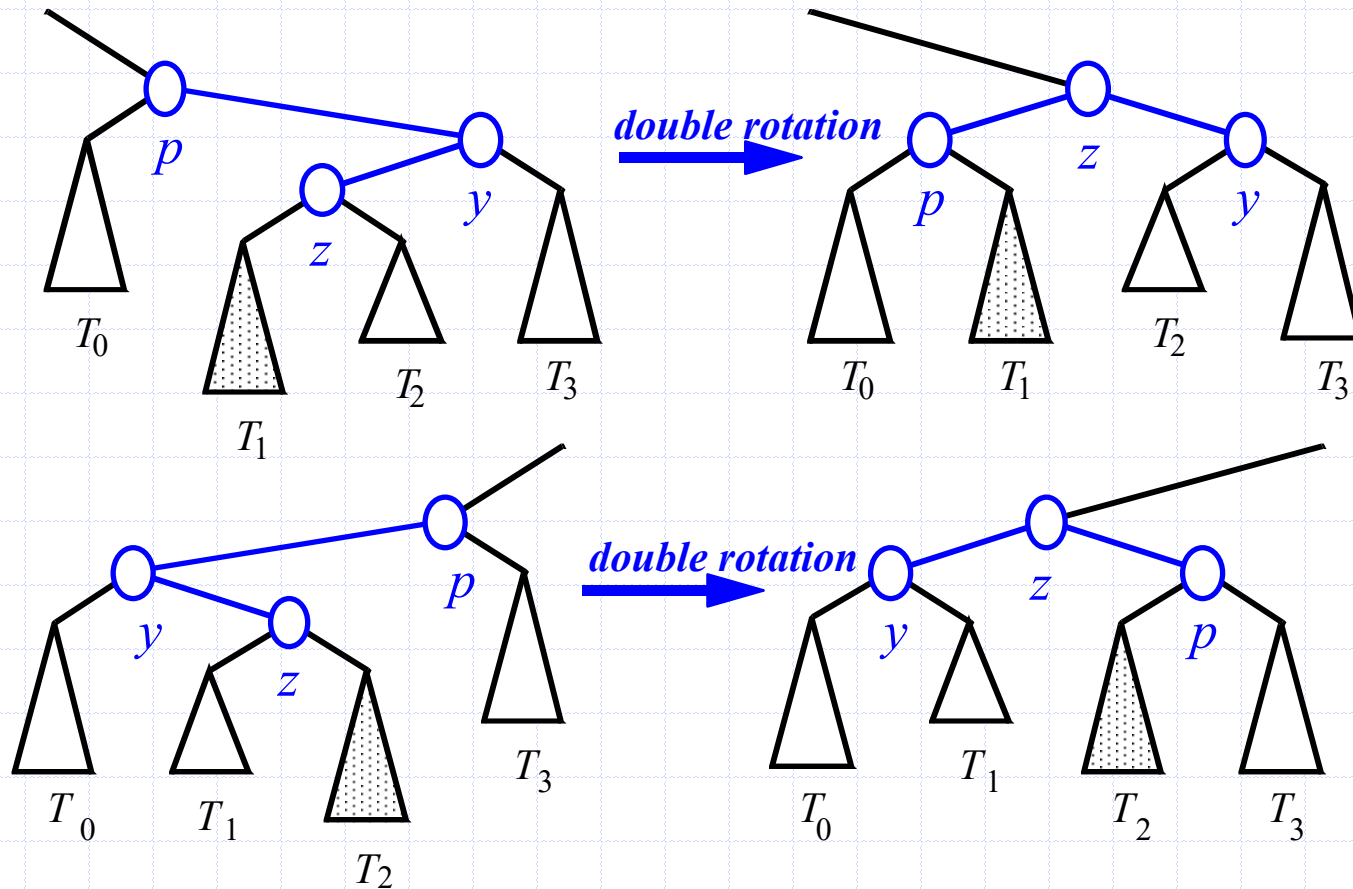
# Restructuring (as Single Rotations)

## ◆ Single Rotations:



# Restructuring (as Double Rotations)

◆ double rotations:



Right  
Left

Left  
Right

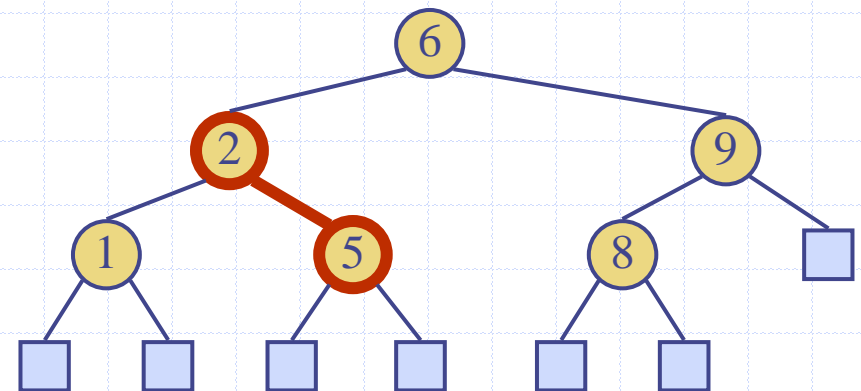
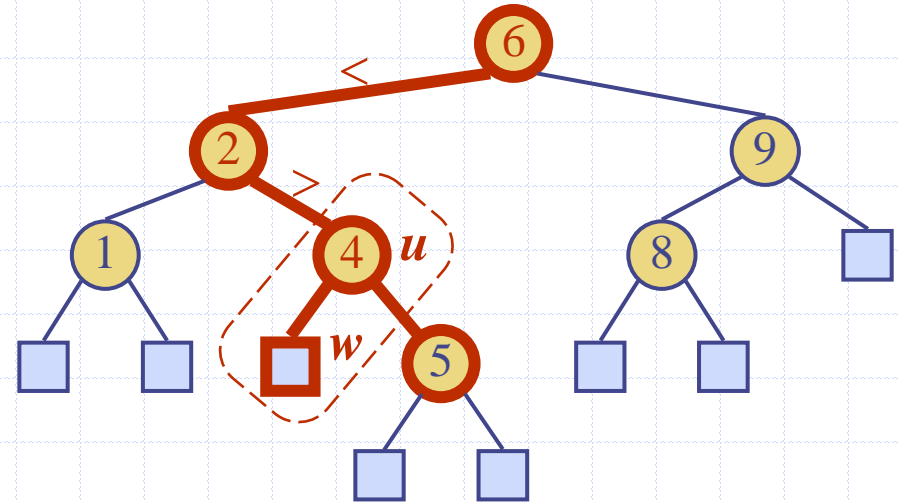
# Red-Black Tree Deletion



# Recall

## Binary Tree Deletion (Case 1)

- ◆ To perform operation **remove( $k$ )**, first search for key  $k$
- ◆ Assume key  $k$  is in the tree, and let  $u$  be the node storing  $k$
- ◆ Two cases:
  1. Node  $u$  has a leaf child  $w$
  2. Node  $u$  has no leaf child
- ◆ If node  $u$  has a leaf child  $w$ , we remove  $u$  and  $w$  from the tree with operation **remove( $u$ )**
- ◆ Example: remove 4

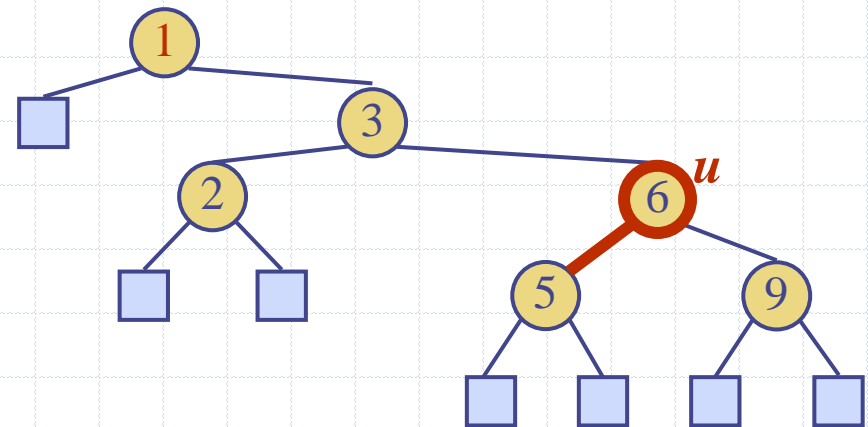
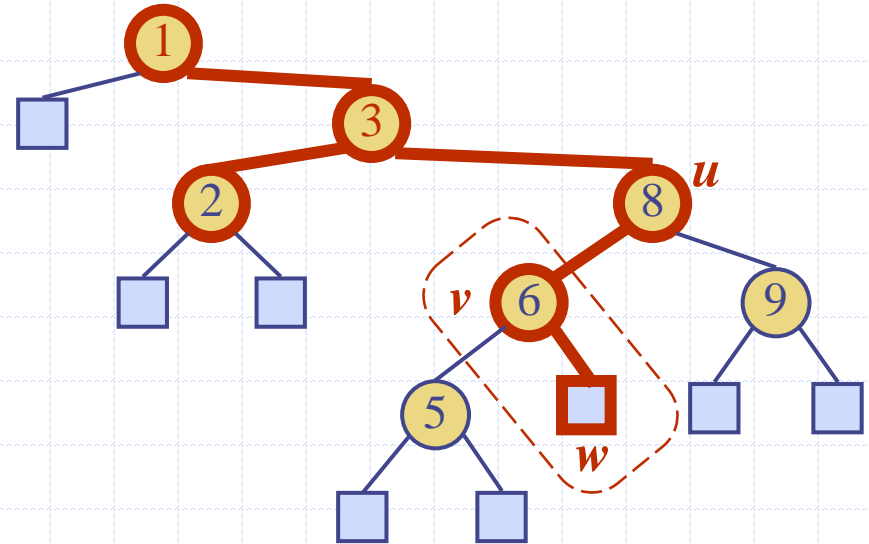


# Recall

## Binary Tree Deletion (Case 2)

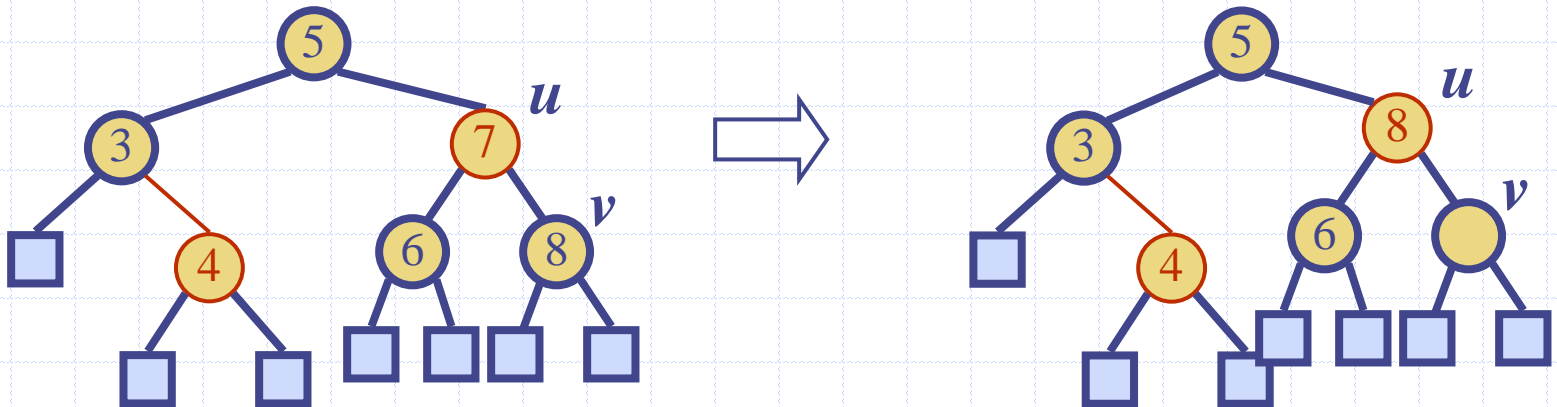
- ◆ We consider the case where the key  $k$  to be removed is stored at a node  $u$  whose children are both internal
  - we find the internal node  $v$  that precedes  $u$  in an in-order traversal ( $v$  in the example)
  - we copy  $key(v)$  into node  $u$
  - we remove node  $v$  and its external child  $w$  by means of operation **remove**( $v$ )

◆ Example: remove 8



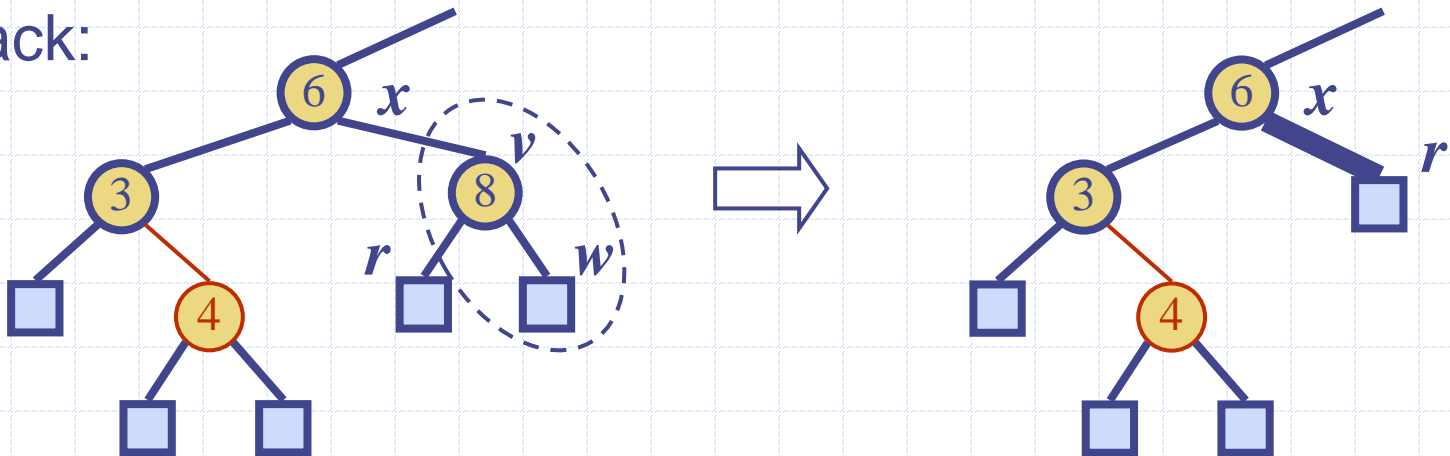
# Deletion

- ◆ To perform operation **remove( $k$ )**, first execute the deletion algorithm for binary search trees
  - If node to be removed,  $u$ , does not have an external child, find next internal node by inorder traversal, called  $v$ , and move key at  $v$  to  $u$ , then remove  $v$ .
  - Thus, every removal occurs at an internal node with an external child.



# Deletion

- ◆ 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
  - Else ( $v$  and  $r$  were both black), so we color  $r$  **double black** (a fictitious color), which is a violation of the internal property requiring a reorganization of the tree (denotes underflow)
- ◆ For example, the deletion of 8 causes a double black:



# Remedying a Double Black

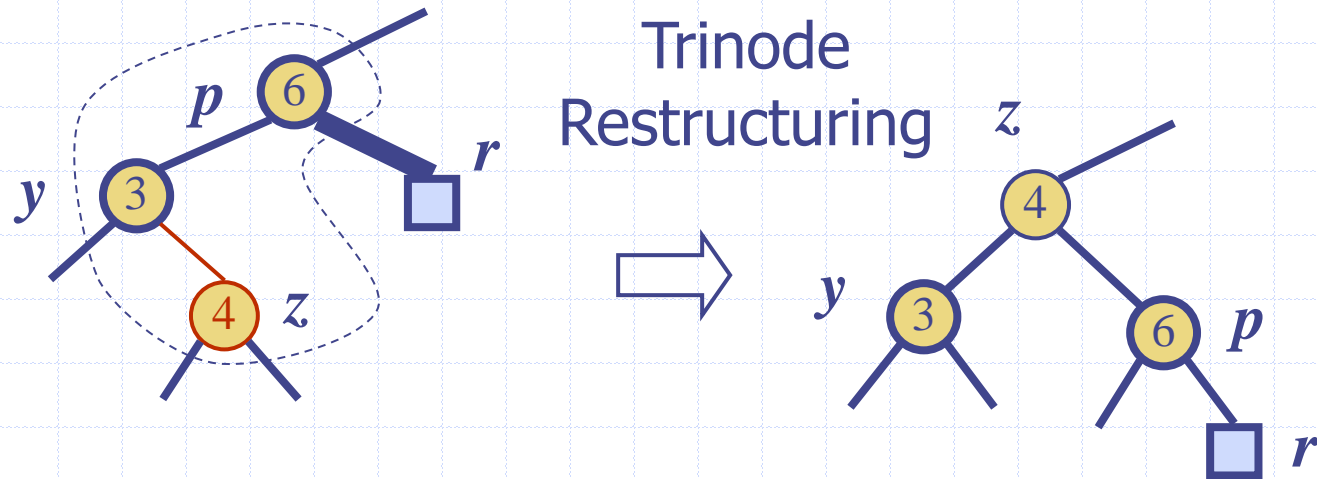
- ◆ The algorithm for remedying a double black node  $r$  with sibling  $y$  considers three cases

Case 1: sibling  $y$  is black and has a red child  $z$

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

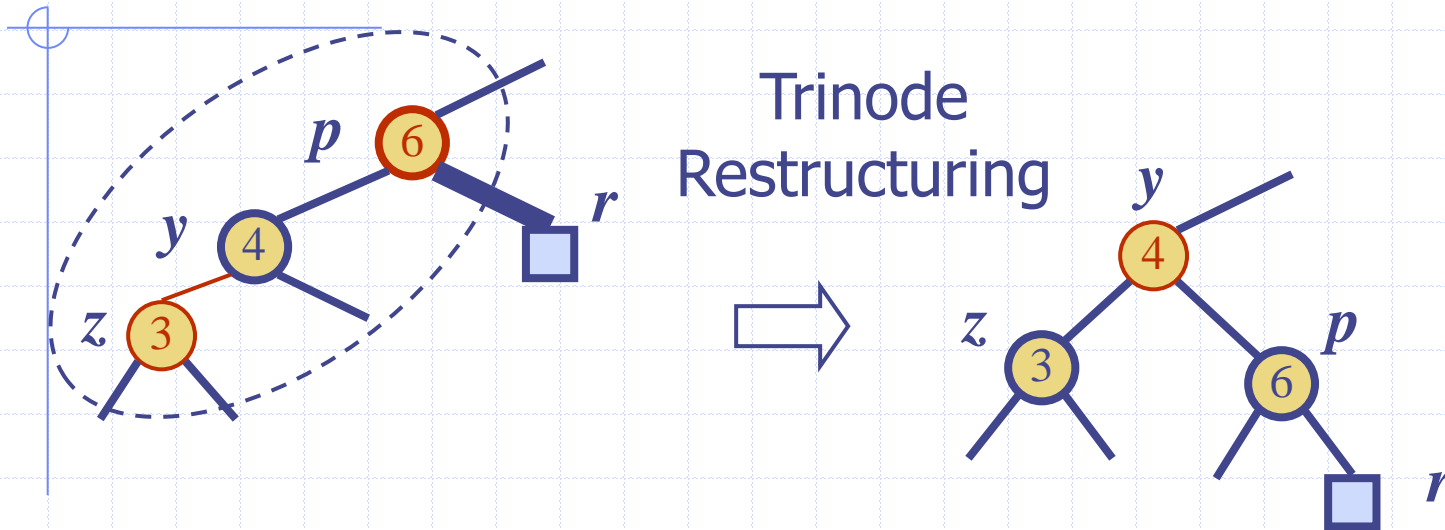
Case 3: sibling  $y$  is red

# Case 1a – sibling $y$ is black and has a red child $z$



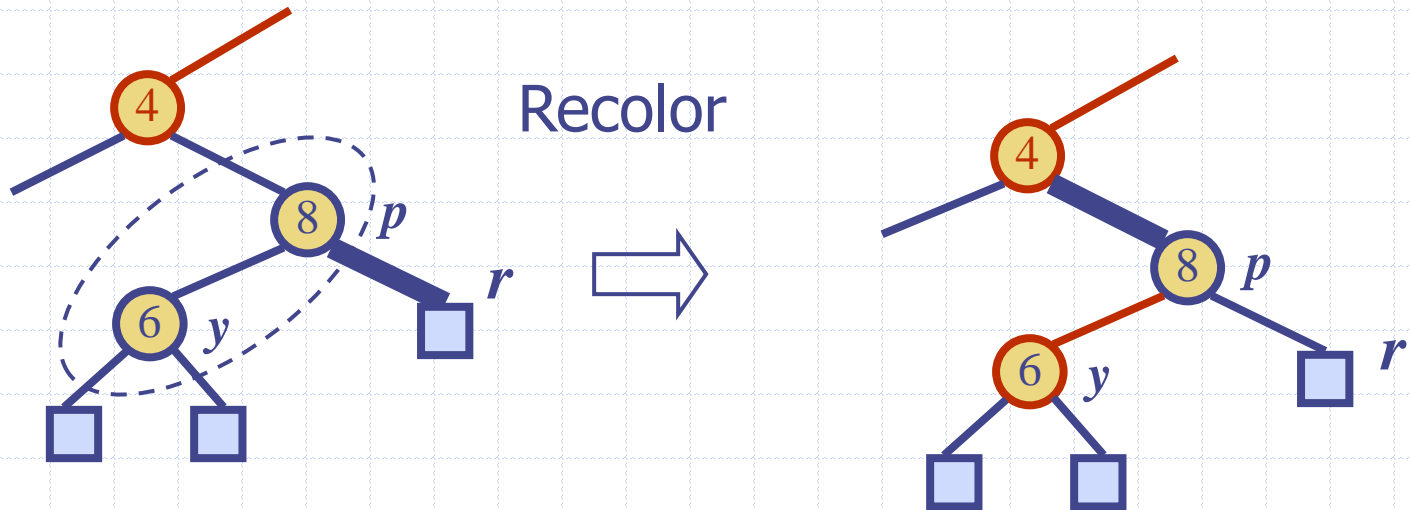
Color  $y$  and  $p$  black, give  $z$  the former color of  $p$ , and set  $r$  to black

# Case 1b – sibling $y$ is black and has a red child $z$



Color  $z$  and  $p$  black, give  $y$  the former color of  $p$ , and set  $r$  to black

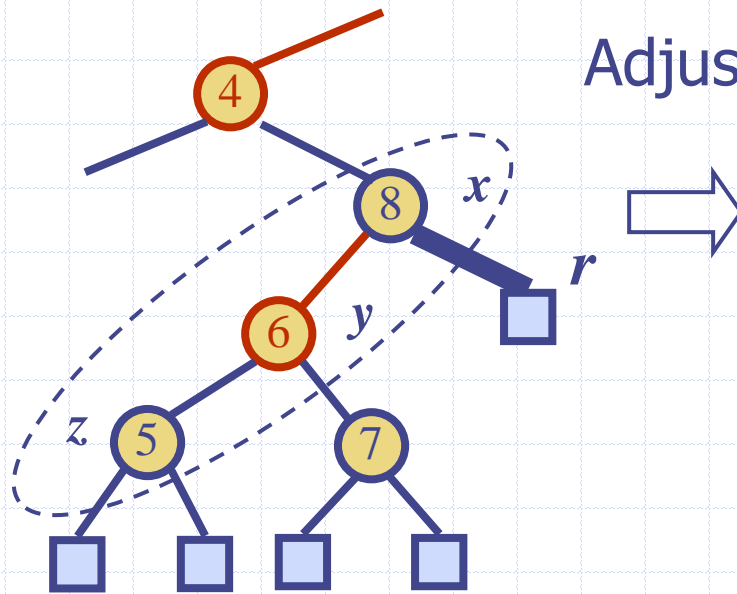
# Case 2 – sibling $y$ is black and its children are both black



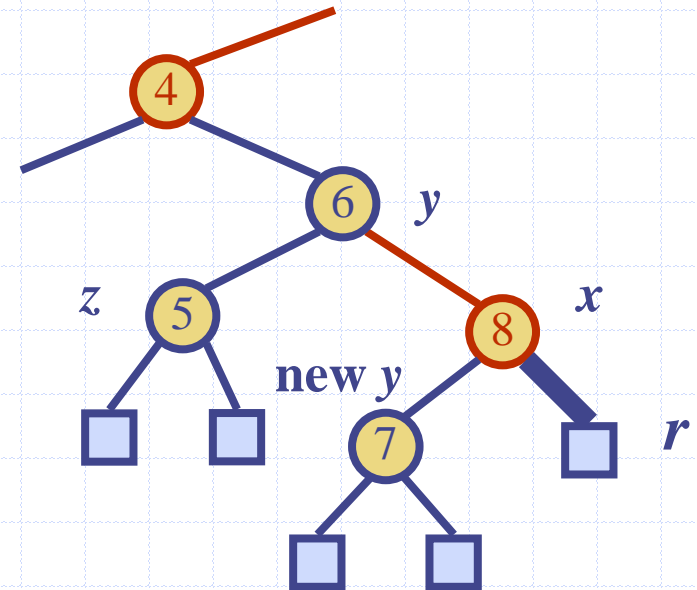
Color  $r$  black and  $y$  red;  
if  $p$  is red, color  $p$  black else  
color  $p$  double black



# Case 3 – sibling y is red



Adjustment



Execute the trinode restructure operation that makes  $y$  the parent of  $x$ ; Color  $y$  black and  $x$  red;

Then apply Case 1 or 2, with new  $y$ .

# Remedying a Double Black

- ◆ The algorithm for remedying a double black node  $r$  with sibling  $y$  considers three cases

**Case 1:** sibling  $y$  is black and has a red child

- Perform a **restructuring**, equivalent to a **transfer**, and we are done

**Case 2:** sibling  $y$  is black and its children are both black

- Perform a **recoloring**, equivalent to a **fusion**, which may propagate the double black violation up to parent

**Case 3:** sibling  $y$  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

- ◆ Deletion in a red-black tree takes  $O(\log n)$  time

# Analysis of Deletion (abstract)

## Algorithm *removeElement(k)*

1.  $v \leftarrow \text{findNode2Remove}(k)$  Search for key  $k$ , then find deletion node  $v$  with external child  $w$ . Let  $y = \text{sibling}(v)$ .
2. *remove*( $v$ ) (removes  $v$  and  $w$  and returns  $r = \text{sibling}(w)$ ). If  $v$  and  $r$  are both black then color  $r$  double black else  $r$  was red so color  $r$  black and we're done
3. **while** *isDoubleBlack*( $r$ )  
    **if** *isRed*( $y$ ) **then**  
         $y \leftarrow \text{adjustment}(y)$   
    **if** *hasRedChild*( $y$ ) **then** //transfer  
        *restructure*( $r$ )  
    **return**  
    **else** {*sibling*( $r$ ) has no red child}  
         $r \leftarrow \text{fusionRecolor}(r)$

- ◆ 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 two restructurings taking  $O(1)$  time each
- ◆ Thus, a deletion in a red-black tree takes  $O(\log n)$  time

# Analysis of Deletion (details)

Algorithm *findNode2Remove(k)*

Input returns node *r* containing key *k*, node *r* has at least one external node

$v \leftarrow \text{findPosition}(k, \text{root}())$

$r \leftarrow v$

if *isInternal(leftChild(v))*  $\wedge$  *isInternal(rightChild(v))* then // one child must be external

$r \leftarrow \text{findPosition}(k, \text{leftChild}(v))$  // finds node containing predecessor of *k*

*swapElements(v, r)* // swaps items so *r* is node containing key being deleted

return *r* // *r* is the node to be deleted and contains key *k* unless *k* is not in tree

Algorithm *fusionRecolor(y, p, r)*

Input *r* and *y* are siblings, *p* is their parent

*setColor(y, RED)*

if *isRed(p)* then

*setColor(p, BLK)*

else

*setColor(p, DOUBLE\_BLACK)*

if *isInternal(r)* then

*setColor(r, BLK)*

return *p*

# Analysis of Deletion (details)

Algorithm *removeDoubleBlack*(*y*, *r*)

Input *r* is the double black node and *y* is *sibling*(*r*)

if *isDoubleBlack*(*r*) then

    if *isRed*(*y*) then {Case 3: when *y*, the sibling of *r*, is red}

*y* ← *adjustment*(*y*)

*p* ← *parent*(*y*)

*z* ← *redChildOf*(*y*)

    if *isBlack*(*z*) then {Case 1: when *y* has no red child}

*r* ← *fusionRecolor*(*y*, *p*, *r*)

        if *isRoot*(*r*) then

*setColor*(*r*, *BLK*)

        else

*removeDoubleBlack*(*sibling*(*r*), *r*) // recursive call

    else {Case 2: *y* has a red child *z*, so we do a transfer}

*restructure*(*z*)

*setColor*(*parent*(*p*), *color*(*p*)) // new parent after restructure

*setColor*(*p*, *BLK*)

*setColor*(*z*, *BLK*)

        if *isInternal*(*r*) then

*setColor*(*r*, *BLK*) // make sure *r* is not external/null

# Main Point

2. Restoring balance after insertion or deletion in a red-black tree only requires a constant number of trinode restructurings (0, 1, or 2) and at most  $O(\log n)$  recolorings. The red-black tree is slightly more complicated than a (2,4) tree because of restructuring, but has a major advantage in space requirements and simplifies splitting and fusion of nodes.

*Science of Consciousness:* The TM technique is a simple, effortless technique that restructures the physiology to a more balanced state.

# Red-Black Tree Reorganization

<b>Insertion</b>		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

<b>Deletion</b>		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

# Main Point

3. A red-black tree is an efficient way to implement an ordered dictionary ADT because it achieves logarithmic worst-case running times for both searching and updating.

*Science of Consciousness:* The TM technique is a very simple, effortless way to facilitate contact with the field of total knowledge, where the fulfillment of intellectual study is achieved, i.e., one feels at home with everything and everyone.



# Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. A (2, 4) tree offers a simple and effective way of maintaining balance in a dynamic tree structure.
2. A red-black tree offers a refinement of the (2, 4) tree by eliminating data slots and optimizing operations.

3. **Transcendental Consciousness** is the unbounded field of pure order and balance and is the basis of order and balance in creation.
4. **Impulses within Transcendental Consciousness**: The dynamic natural laws within this unbounded field create and maintain the order and balance in creation.
5. **Wholeness moving within itself**: In Unity Consciousness, the diversity of creation is experienced as waves of intelligence, perfectly orderly fluctuations of one's own self-referral consciousness.