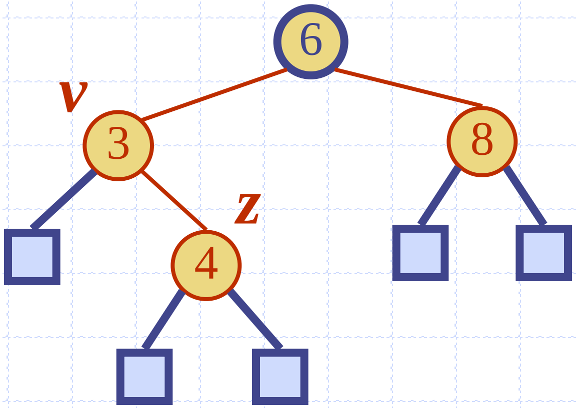
Red-Black Trees



(2,4) Trees

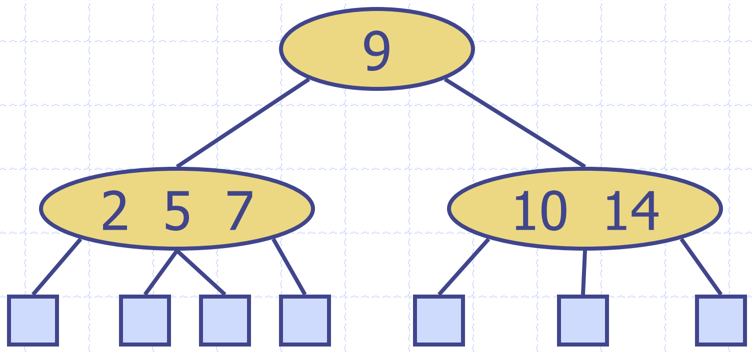
- Good
  - $O(\log n)$ worst case performance for search/insert/delete
- Bad
  - Non-standard trees (i.e., "not binary trees")
  - Implementation complexity

Improvement to (2,4) Trees

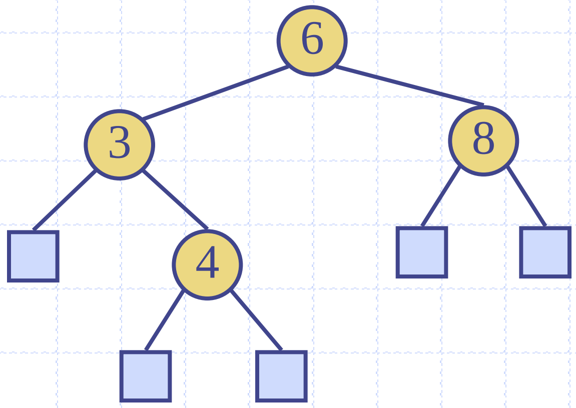
- Currently, we perform constant time "tree-correction" operations that maintain the $O(\log n)$ tree height

- So, can we perform constant time "tree-correction" operations on a standard binary tree and maintain $O(\log n)$ tree height?

Ideas?



"we *have* stuff like this"

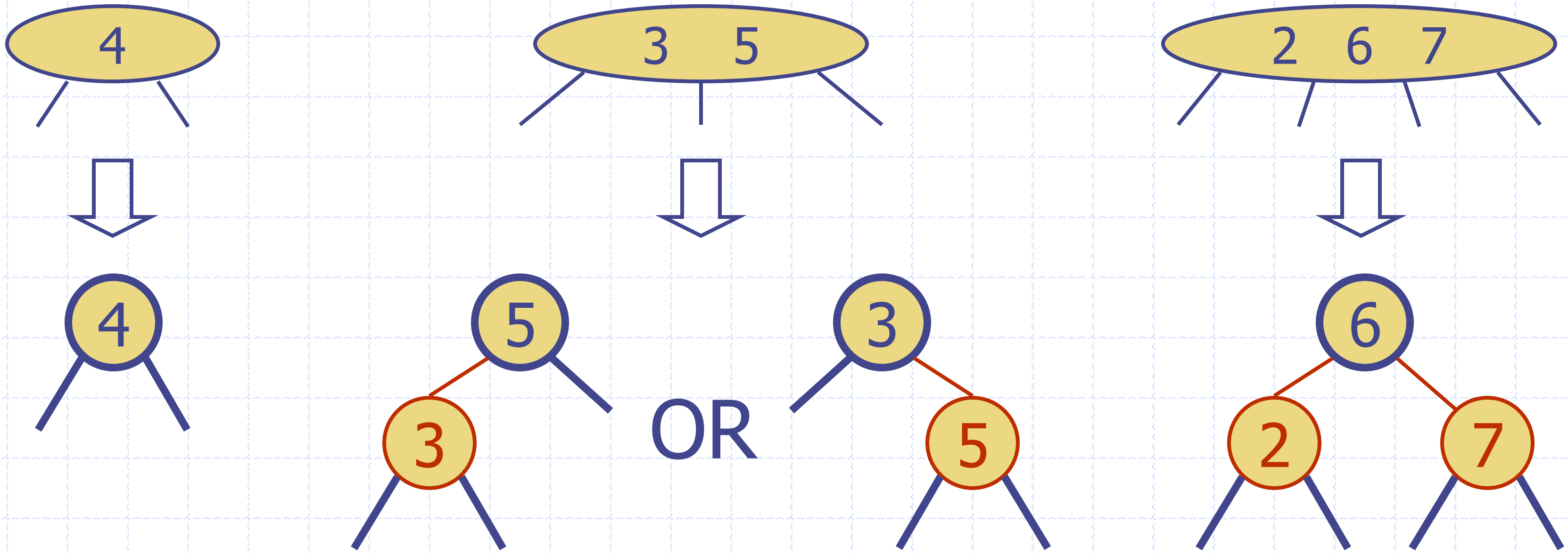


"we *want* stuff like this"

- Welcome to the world of Red-Black trees…

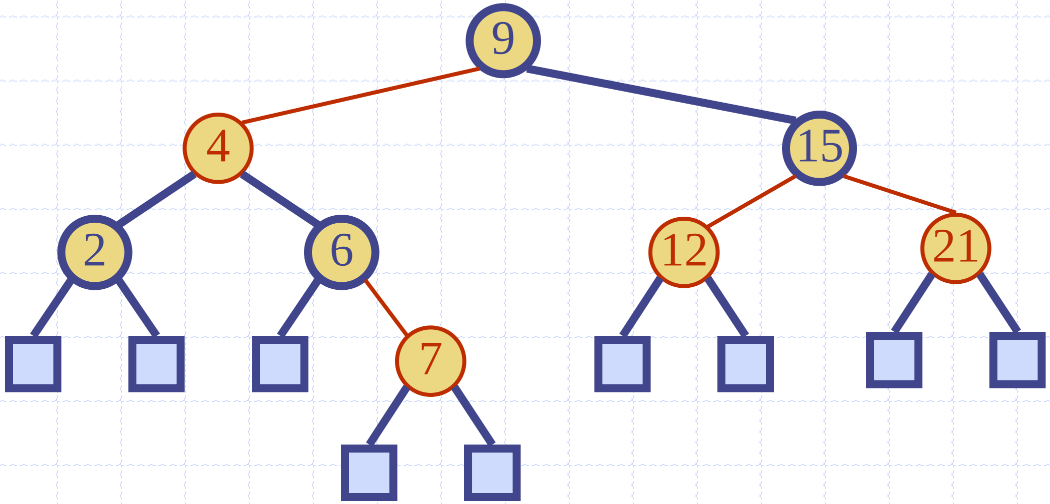
From (2,4) to Red-Black Trees

- 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 (binary-tree-like) 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 leaves have the same black depth**



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 $O(\log n)$
  - (Why?)
- Since a red-black tree is a binary tree, the search algorithm for a red-black 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

Red-Black Tree Operations

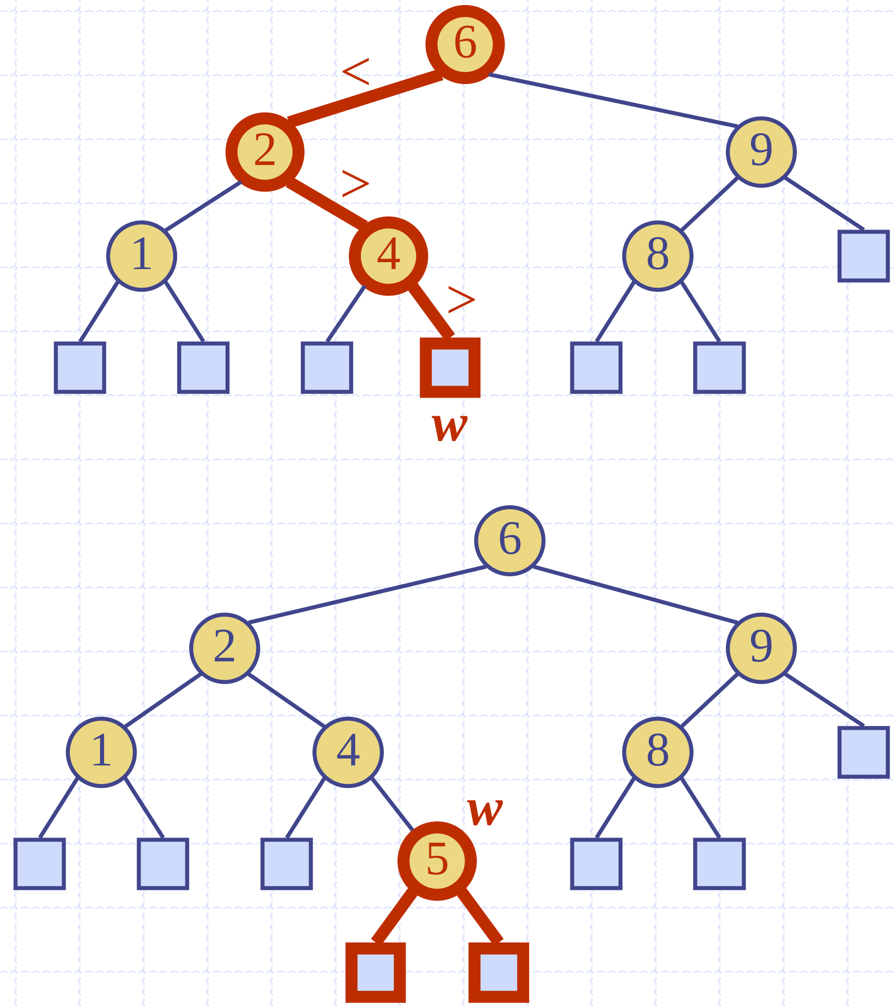
- Search
  - Depends on height of tree, thus searching with n items takes $O(\log n)$
- Insert
  - Coming up next…
- Delete
  - Coming up next next…

Insertion

- To perform operation insertItem(k, o), we execute the insertion algorithm for binary search trees
- ..and color red the newly inserted node z unless it is the root

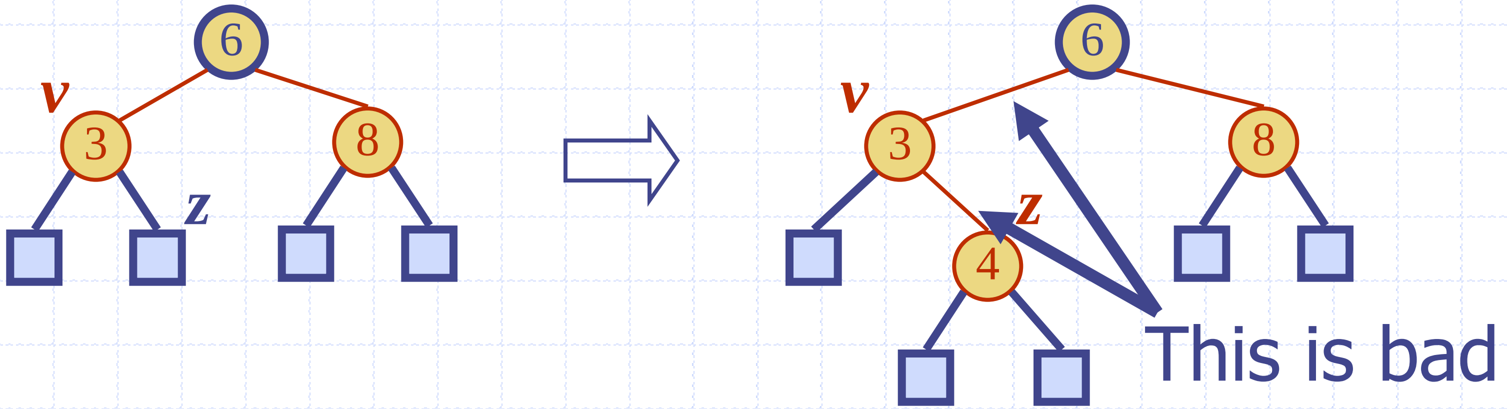
(Insertion for binary trees)

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



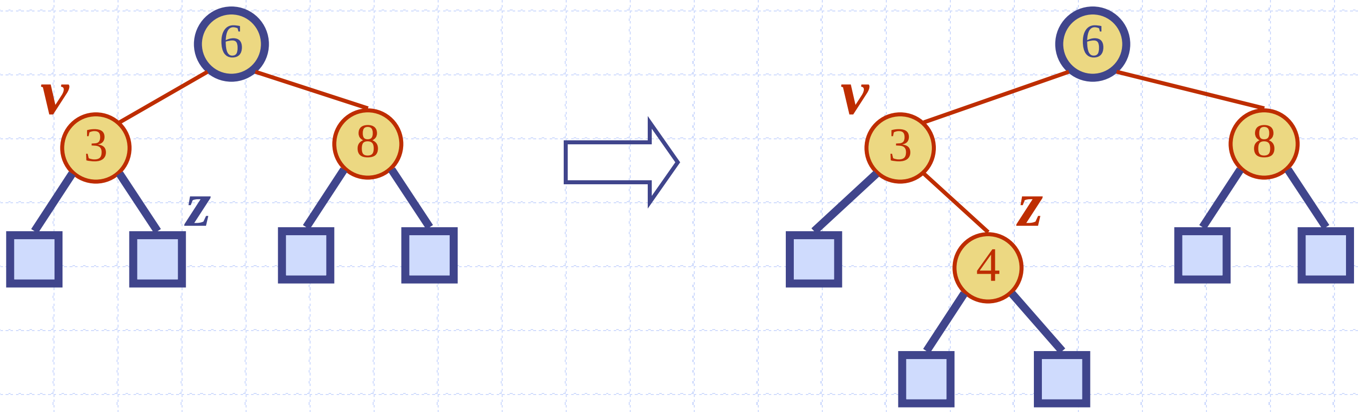
Insertion

- To perform operation insertItem(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:



What can we do?

- Example where the insertion of 4 causes a double red:

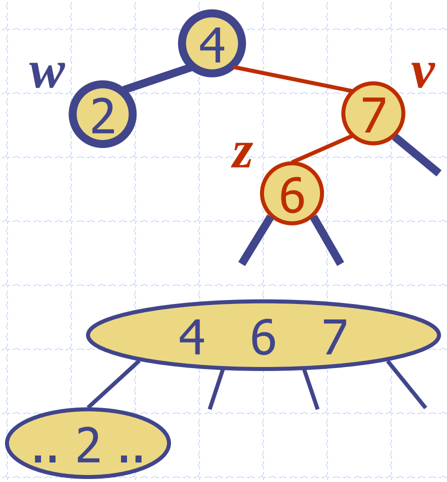


Remedying a Double Red

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

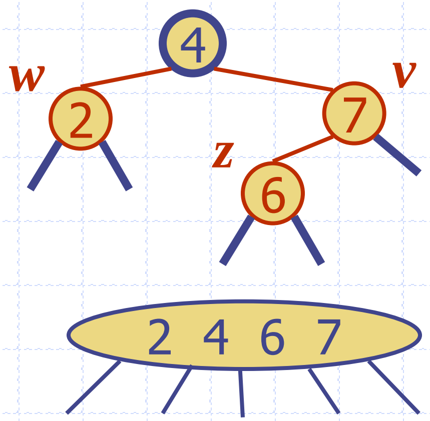
Case 1: w is black

- The double red is an incorrect replacement of a 4-node
- Solution:
  - we change the 4-node replacement = “restructuring”



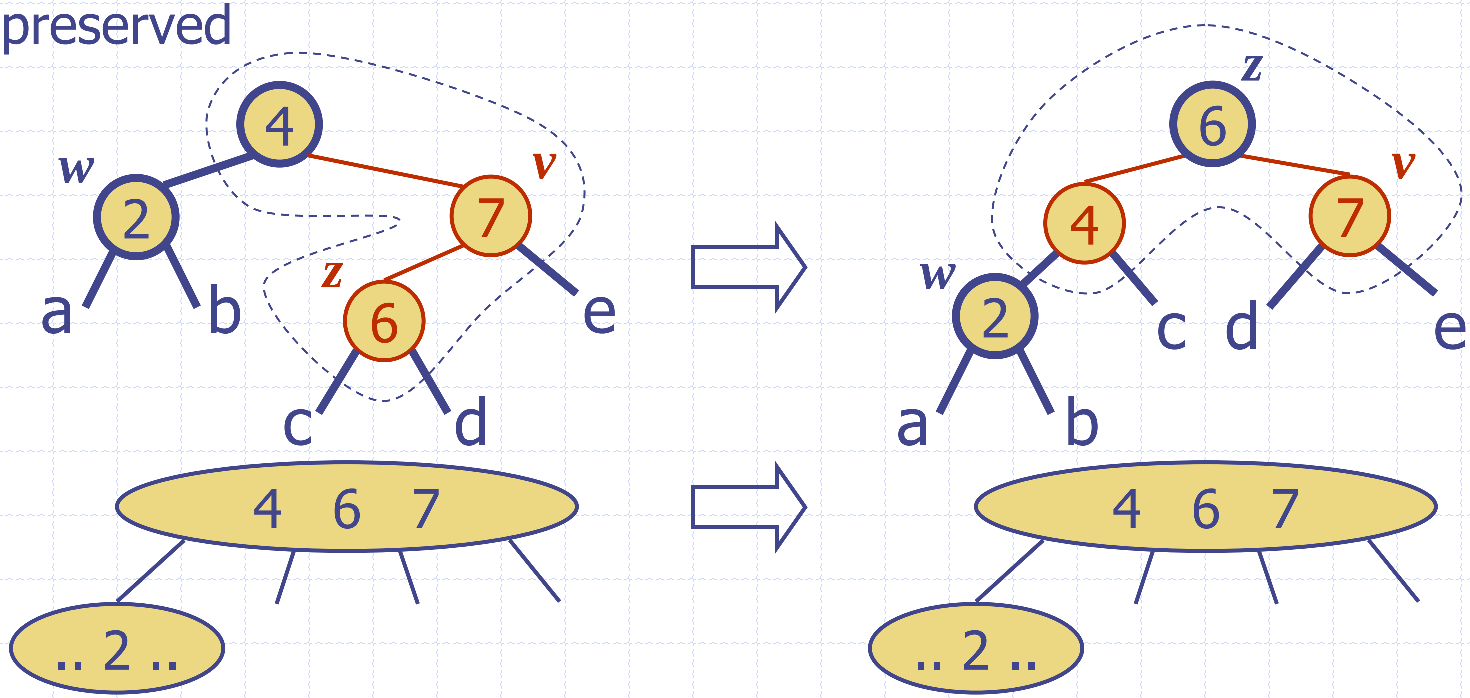
Case 2: w is red

- The double red corresponds to an *overflow*
- Solution:
  - we perform the equivalent of a split = “recoloring”



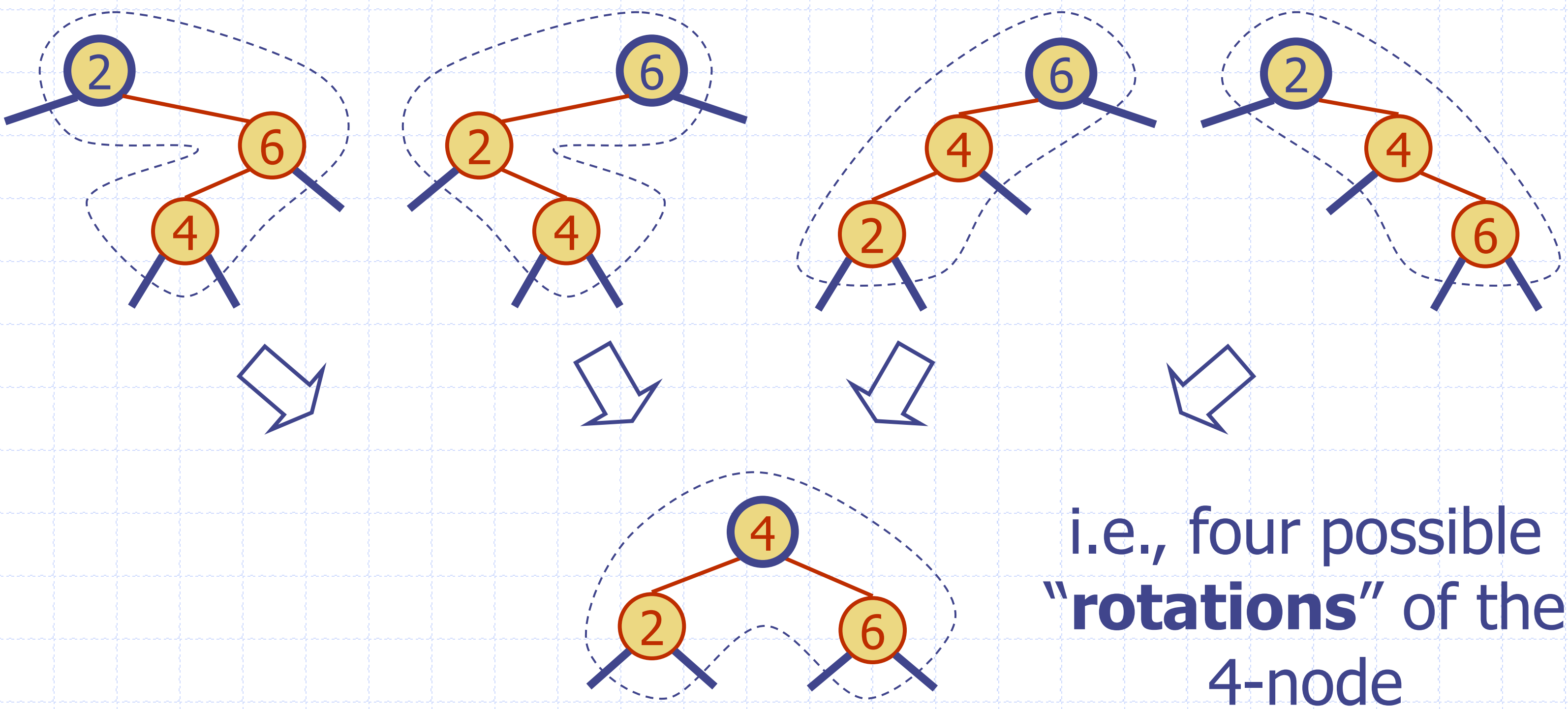
Restructuring

- 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 several restructuring configurations depending on whether the double red nodes are left or right children
  - How many?



i.e., four possible "**rotations**" of the 4-node

Restructuring (cont.)

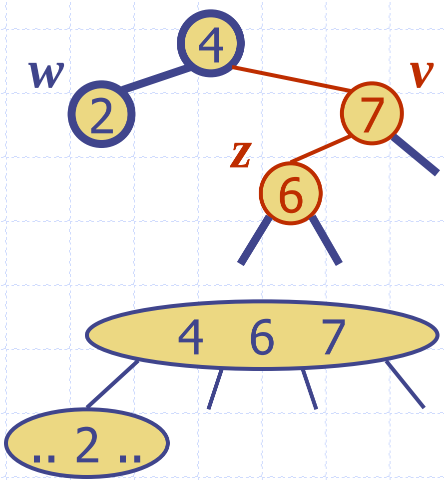
- Note: sometimes restructuring operations are refered to as “rotation operations”

Remedying a Double Red

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

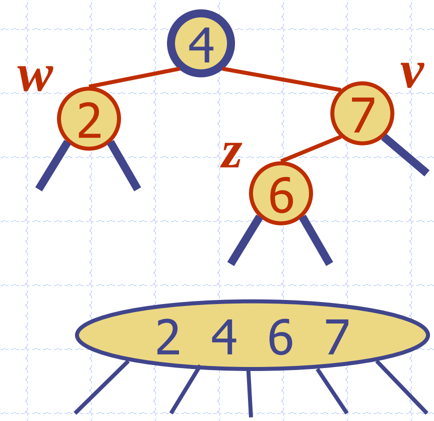
Case 1: w is black

- The double red is an incorrect replacement of a 4-node
- Solution:
  - we change the 4-node replacement = "restructuring"



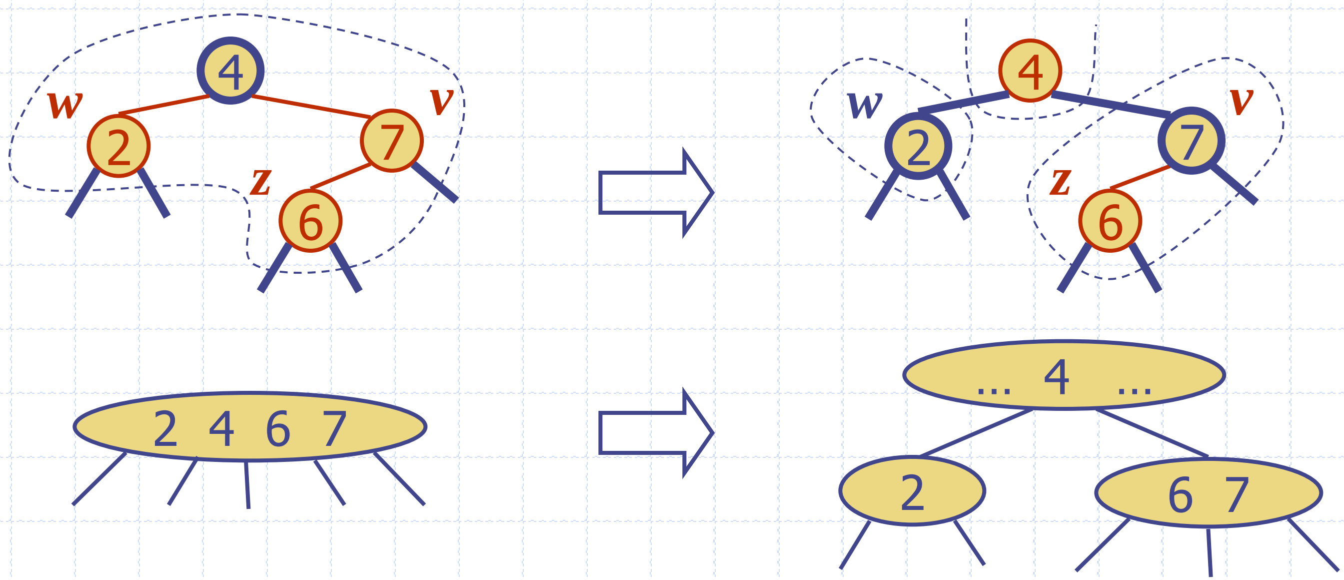
Case 2: w is red

- The double red corresponds to an *overflow*
- Solution:
  - we perform the equivalent of a split = "recoloring"



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***



Analysis of Insertion

Algorithm *insertItem*(k, o)

1. We search for key k to locate the insertion node z

2. We add the new item (k, o) at node z and color z red

3. **while** *doubleRed*(z)
   if *isBlack*(*sibling*(*parent*(z)))
     $z \leftarrow$ *restructure*(z)
     return
   else { *sibling*(*parent*(z)) is red }
     $z \leftarrow$ *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

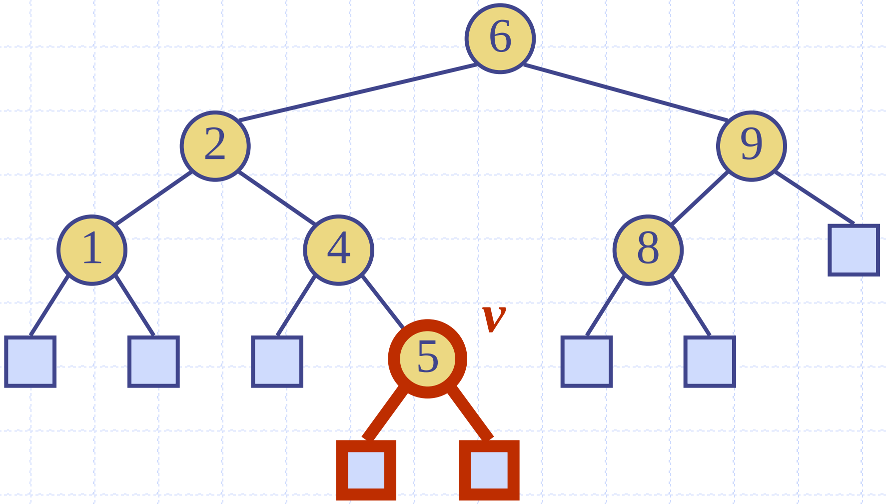
- To perform operation remove(k), we first execute the deletion algorithm for binary search trees

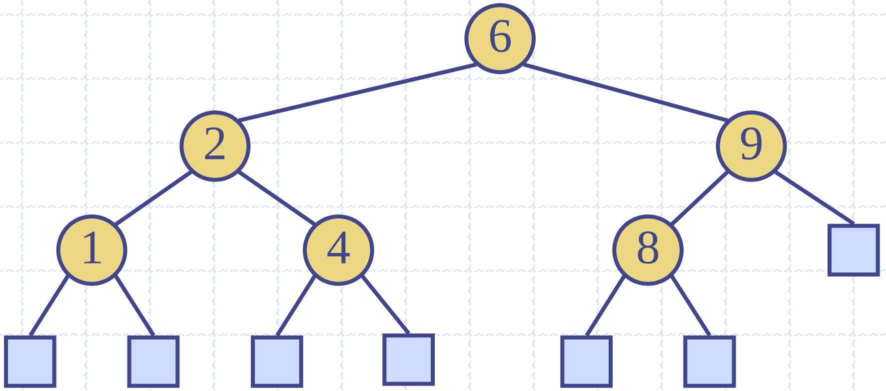
(Deletion for binary trees)

- Three cases:
  - Zero children
  - One child
  - Two children

(Deletion: zero children)

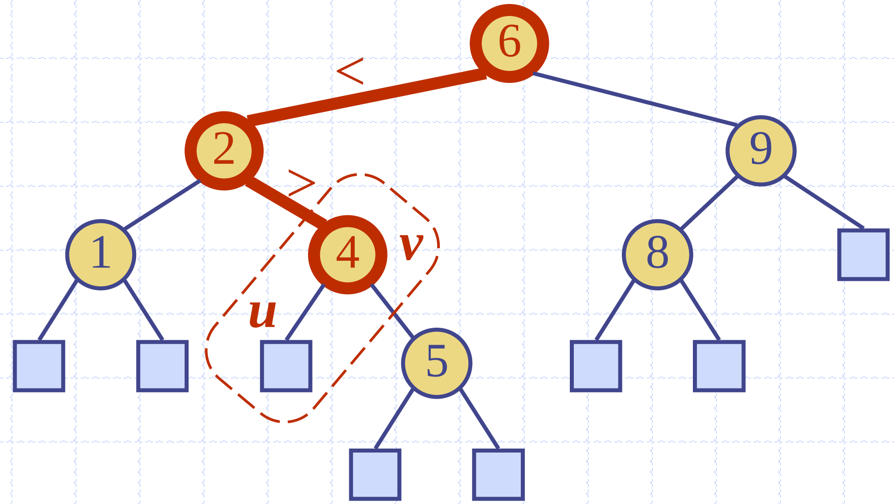
- Must be a leaf node – simple (e.g., remove 5)
  - Assume key ***k*** is in tree, and let ***v*** be the node storing ***k***
  - We search for key ***k***
  - Remove node

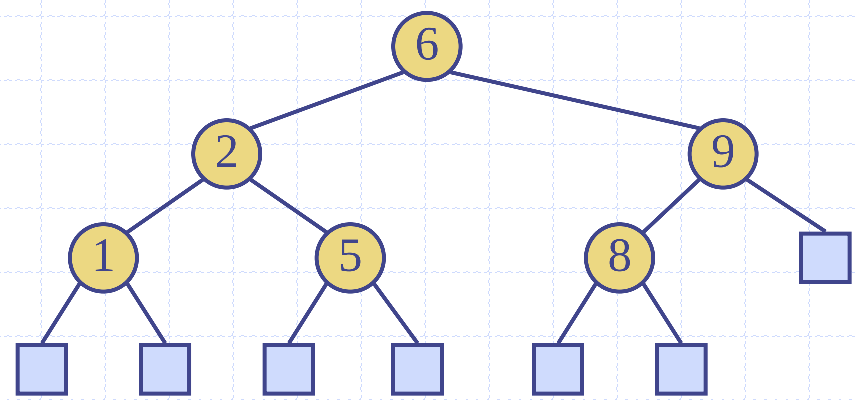




(Deletion: one child)

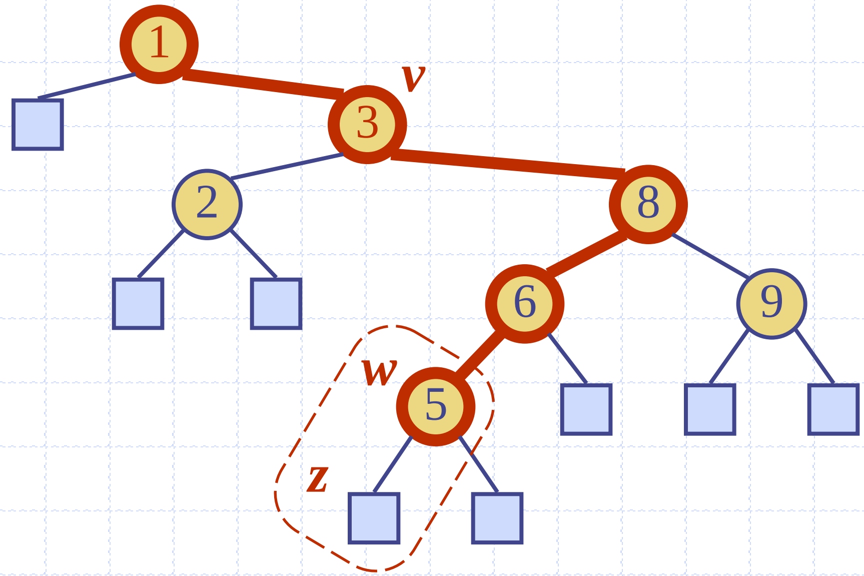
- To perform operation, we search for key k (e.g., remove 4)
- Assume key k is in tree, and let v be the node storing k
- If node v has **one** leaf child u, we remove v and u from the tree with operation removeAboveExternal(u)

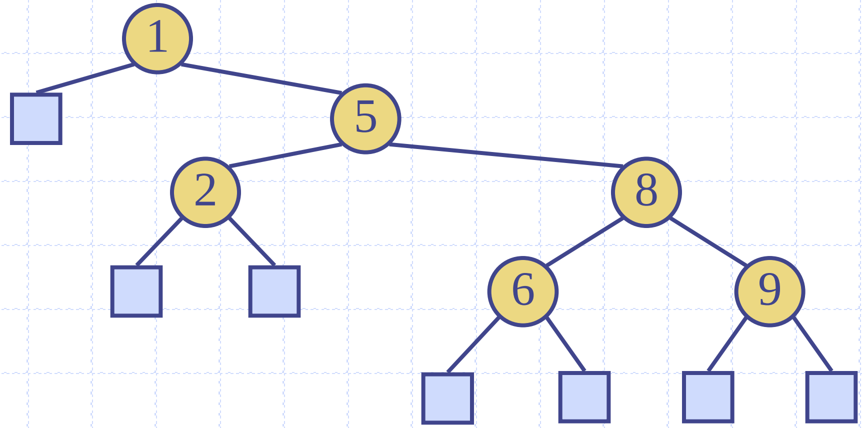




(Deletion: two children)

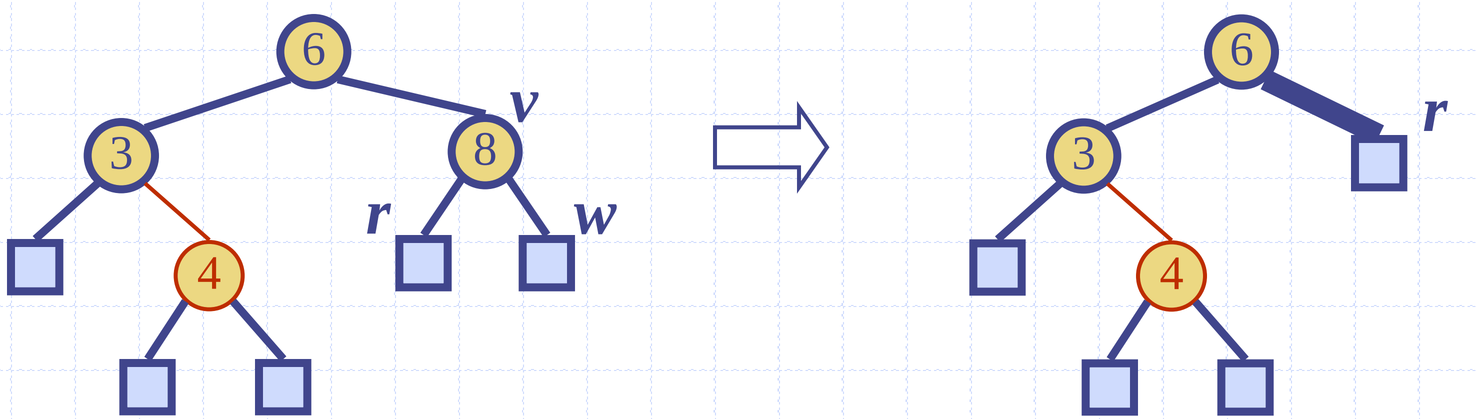
- What if the key k to be removed has **two** internal nodes as children, e.g. "remove 3"
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)





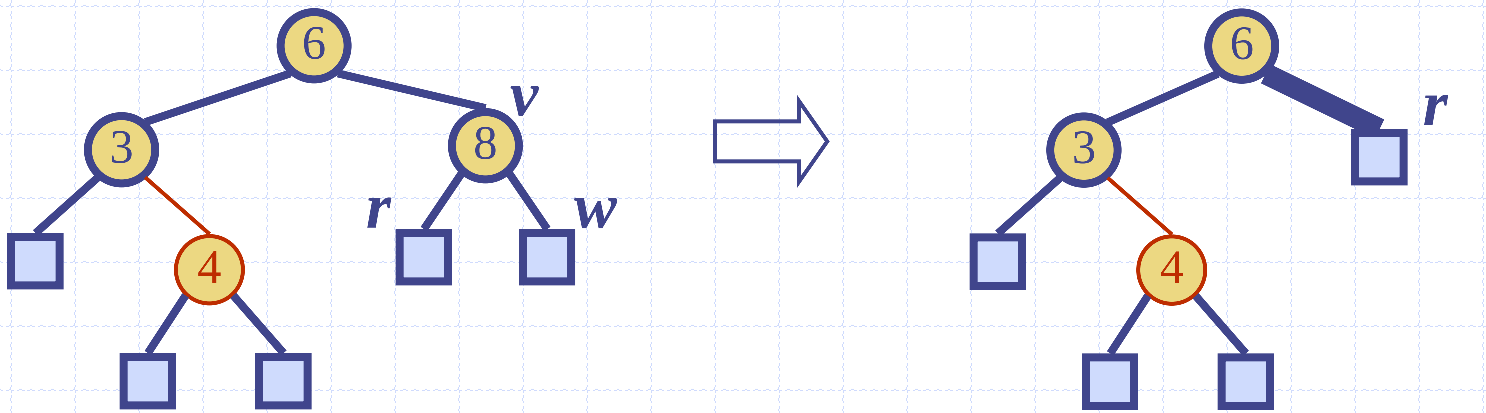
Deletion

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees
- 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) we color r ***double black***, which is a violation of the internal property requiring a reorganization of the tree
- Example where the deletion of 8 causes a double black:



What can we do?

- Example where the deletion of 8 causes a double black:



Remedying a Double Black

- The algorithm for remedying a double black node 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

  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

  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
- 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

Demo

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html