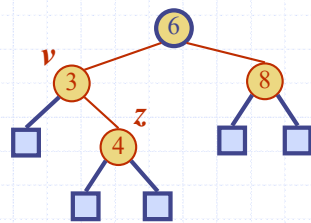


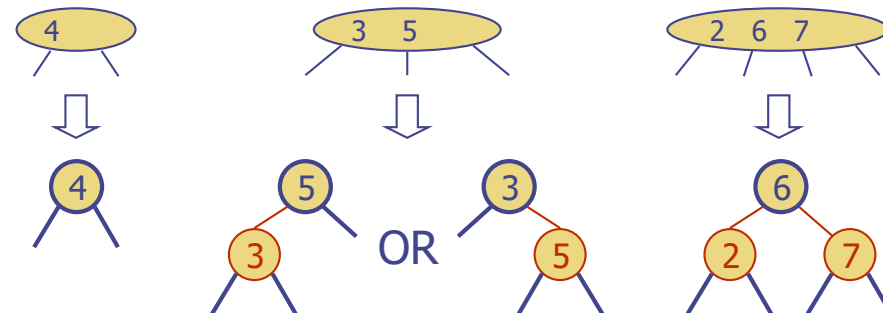
Red-Black Trees



1

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

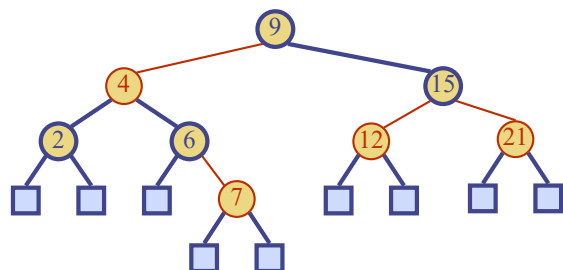


2

Red-Black Trees

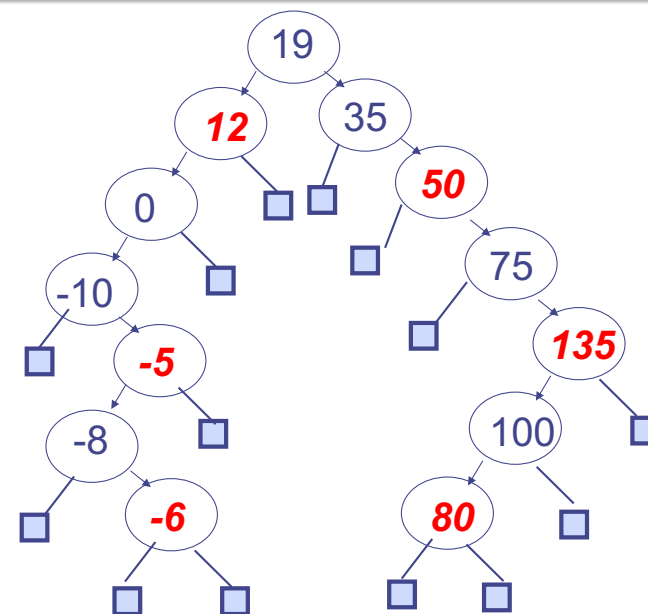
- ◆ A red-black tree can also be defined as a binary search tree that satisfies the following properties:

- **Root Property:** the root is black
- **External Property:** every leaf is black
- **Internal Property:** the children of a red node are black (red rule)
- **Depth Property:** all the leaves have the same black depth (path rule)
- (Question) How is balancing enforced here?



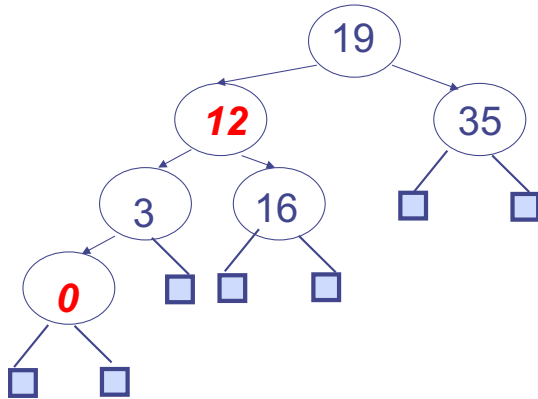
3

Red Black Tree?



4

Red Back Tree?



What if we attach a child to node 0?

5

Implications

- ◆ **Root Property:** the root is black
 - ◆ **External Property:** every leaf is black
 - ◆ **Internal Property:** the children of a red node are black (red rule)
 - ◆ **Depth Property:** all the leaves have the same black depth (path rule)
-
- ◆ 1. If a red node has any children, it must have two children and they must be black
 - Why? Depth property
 - ◆ 2. If a black node has only one “real” child then it must be a “last” red node
 - If the child is black?
 - If the child is not the last red?
-
- ◆ (Question) How is balancing enforced in R-B tree?

6

Intuition about “rough balancing”

- ◆ The longest path $\leq 2 \times$ the shortest path
 - Rough balancing \rightarrow guarantees $\log(n)$ height
- ◆ Why?
 - From “red rule” and “path rule”
shortest path = only black nodes
longest path = inserting a red node between two black nodes

Root Property: the root is black
External Property: every leaf is black
Internal Property: the children of a red node are black (red rule)
Depth Property: all the leaves have the same black depth (path rule)

7

Height of a Red-Black Tree

- ◆ **Theorem:** A red-black tree storing n entries has height $O(\log n)$
Proof:
 - Omitted
- ◆ 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

8

Insertion

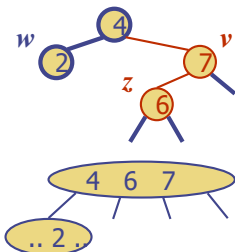
9

Remedying a Double Red

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

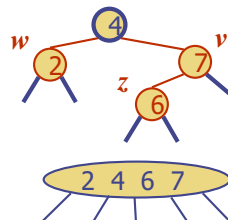
Case 1: w is black

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



Case 2: w is red

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

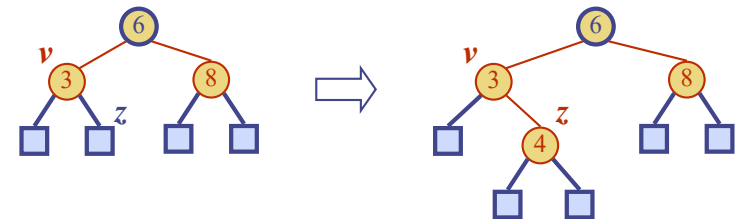


11

Insertion

- ◆ To perform operation **put(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
 - Goal: Removing double red without breaking the depth property

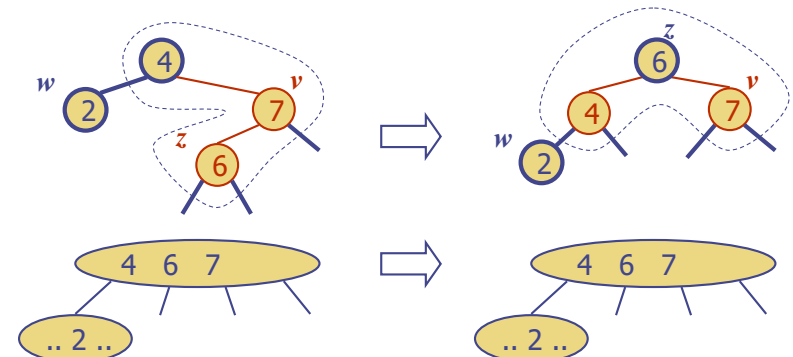
- ◆ Example where the insertion of 4 causes a double red:



10

Restructuring

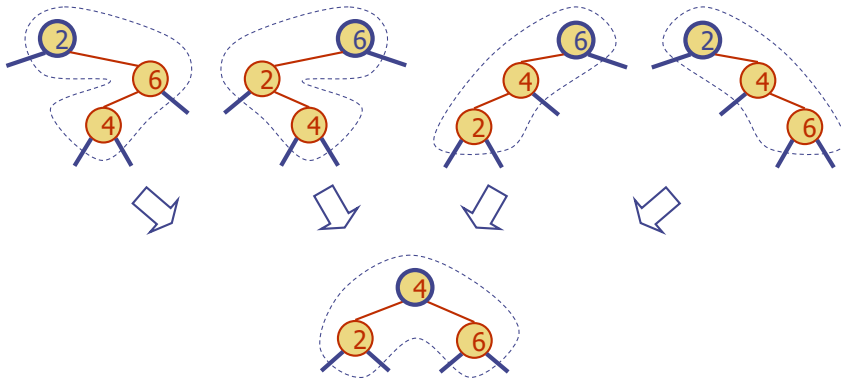
- ◆ A restructuring remedies a child-parent double red when the parent red node has a black sibling
- ◆ It is equivalent to restoring the correct replacement of a 4-node
- ◆ The internal property is restored and the other properties are preserved



12

Restructuring (cont.)

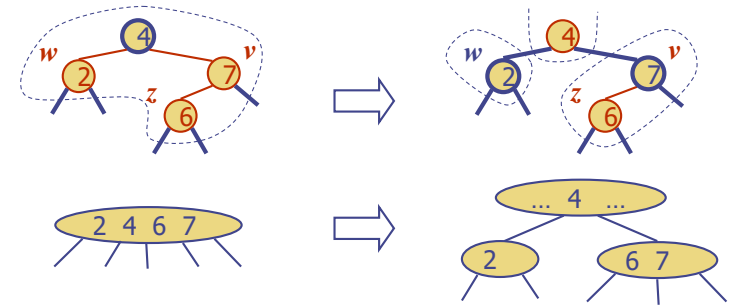
- There are four restructuring configurations depending on whether the double red nodes are left or right children



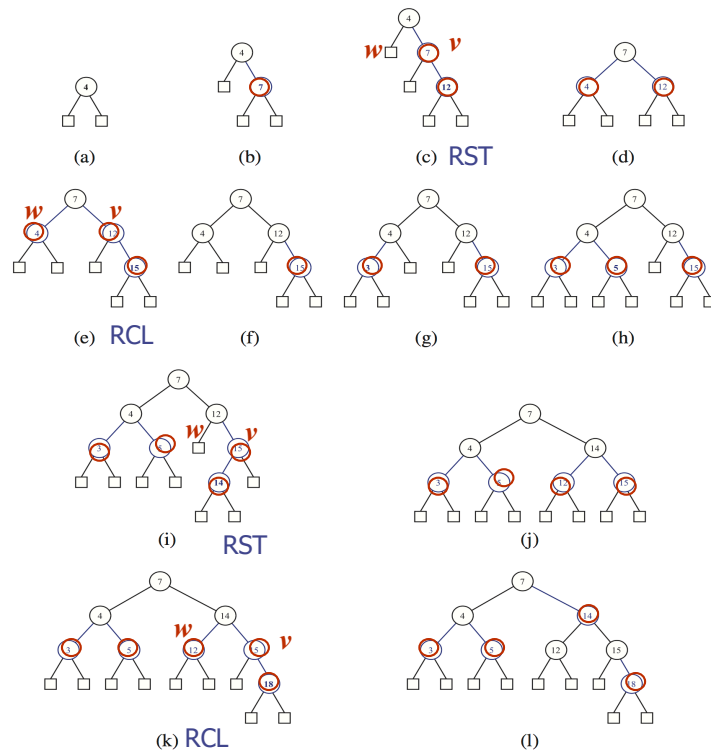
13

Recoloring

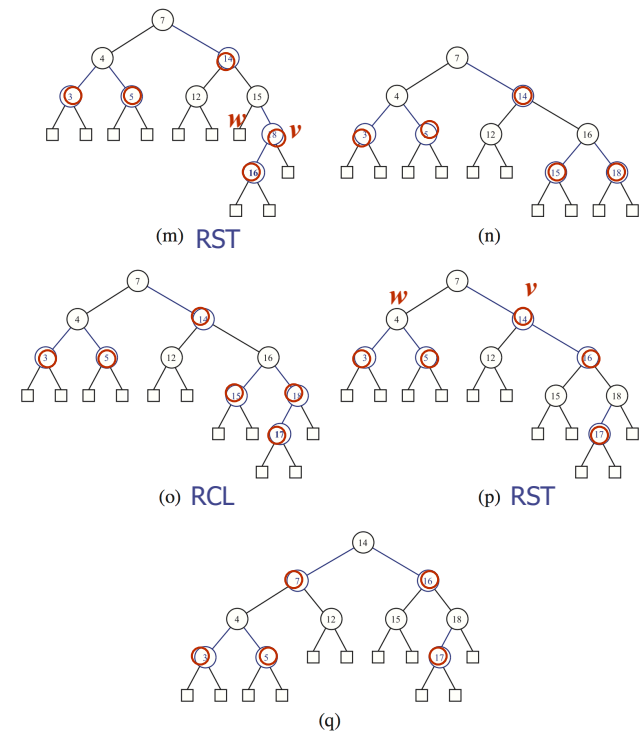
- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate to the grandparent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



14



15



16

Analysis of Insertion

Algorithm *put(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)))
 $z \leftarrow \text{restructure}(z)$
 return
 else { *sibling*(*parent*(z)) is red }
 $z \leftarrow \text{recolor}(z)$

- ◆ 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

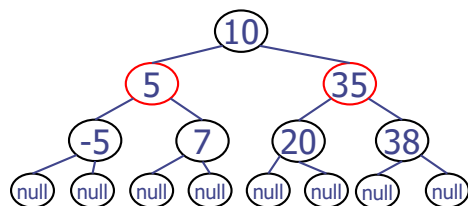
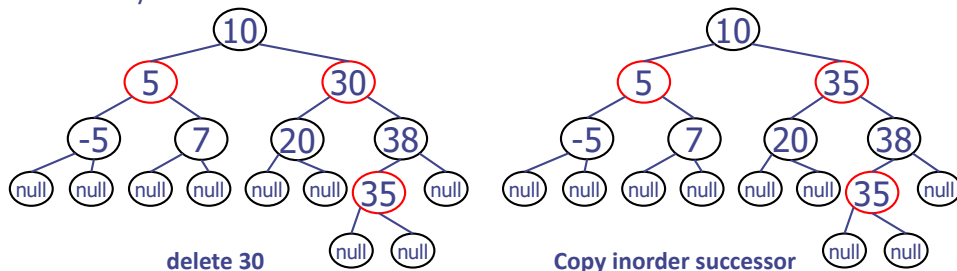
17

RB-Tree: Deletion

18

Deletion: Example 1

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



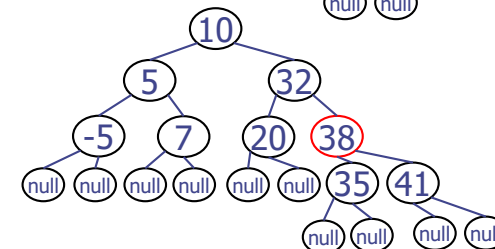
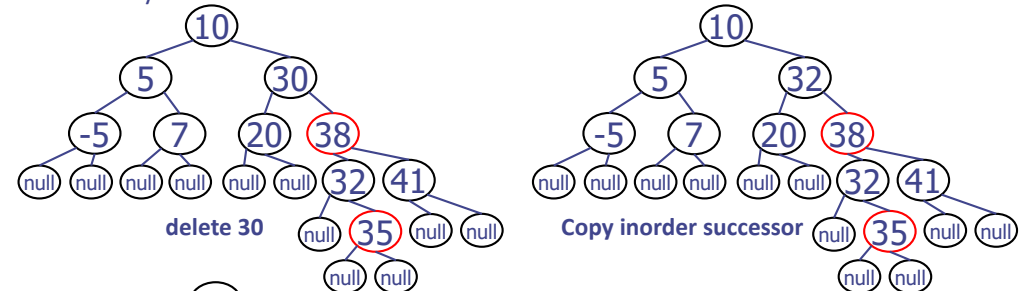
Just delete the copied 35, and color the remaining node in black. Then, we are done.

Implication:
If the node to be deleted is red,
 removing it is fine

19

Deletion: Example 2

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



Just delete the copied 32, and color 35 with black.

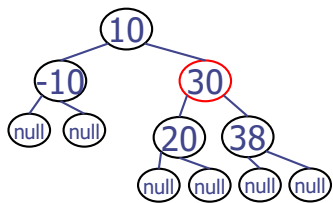
Implication: **For a node (with a red child) to be deleted,** delete it and change the red child's color.

(35: -1 first and +1 second. So no change)

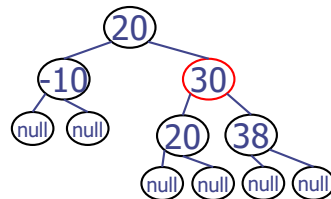
20

Deletion: Example 3

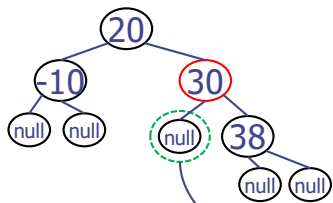
- What about deleting a node with a black child?



Delete 10



Copy inorder successor



Delete 20.

Problem: A path of only 2 blacks

Regard this as "double black nodes"

21

Questions

- How to handle "double black nodes"
- Are there some cases in handling those? Yes
- Are you ready for "cases"?
- It's really, really complex, but if you concentrate, then you can follow it.

23

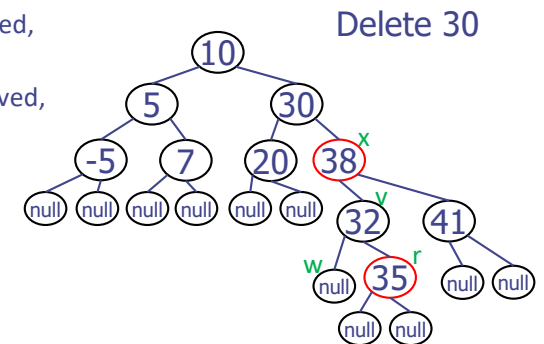
Deletion

- To perform operation $\text{erase}(k)$, we first execute the deletion algorithm for binary search trees

- Enough to consider the removal of an entry at a node with an external child (To remove a node with both internal children, we first copy the inorder successor, and then ...)

Notations

- v : the internal node removed,
 - "myself"
- w : the external node removed,
 - "my lonely child"
- r : the sibling of w
 - "my other child"
- x : the parent of v
 - "my father"



Delete 30

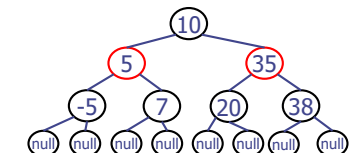
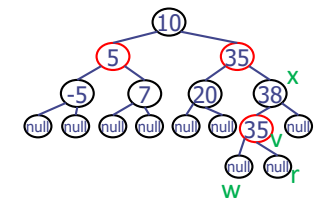
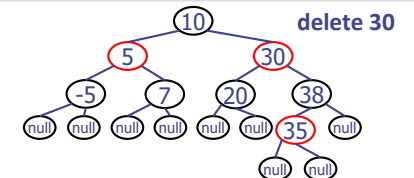
22

Deletion: Algorithm Overview (1)

First, remove v and w , and make r a child of x

If either of v or r was red, we color r black and we are done (Examples 1 and 2)

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 (Examples 3)



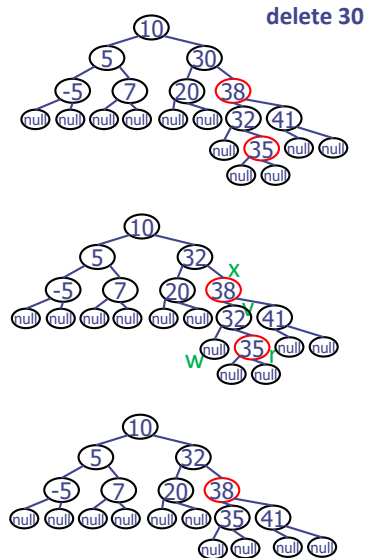
24

Deletion: Algorithm Overview (2)

First, remove v and w , and make r a child of x

If either of v or r was red, we color r black and we are done (Examples 1 and 2)

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 (Examples 3)



25

Deletion: Algorithm Overview (2)

First, remove v and w , and make r a child of x

If either of v or r was red, we color r black and we are done (Examples 1 and 2)
(Let's call this Case 0)

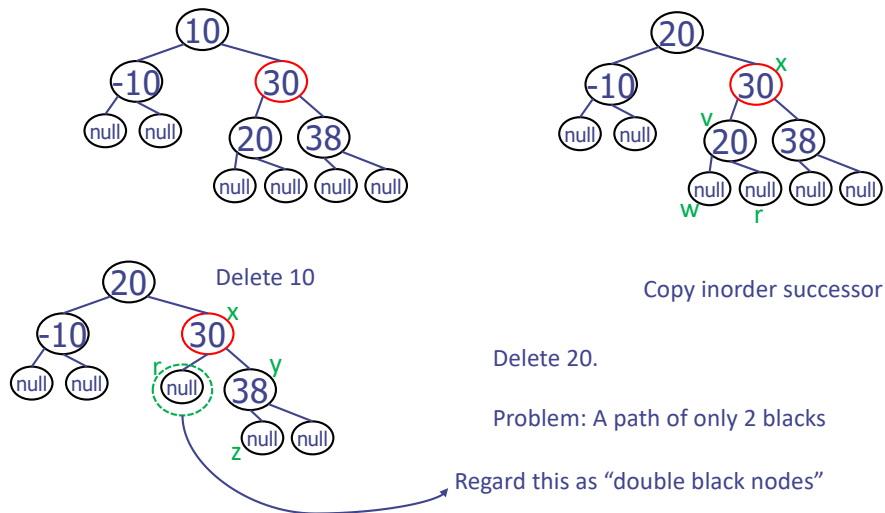
Else (v and r were both black) we color r **double black**, which is a violation of the internal property requiring a reorganization of the tree (Examples 3)

- Notations after removing v and w
 - y : sibling of r
 - z : child of y
- We now divide the cases, depending of the color of y and z

26

Recall: Example 3. Notations again!

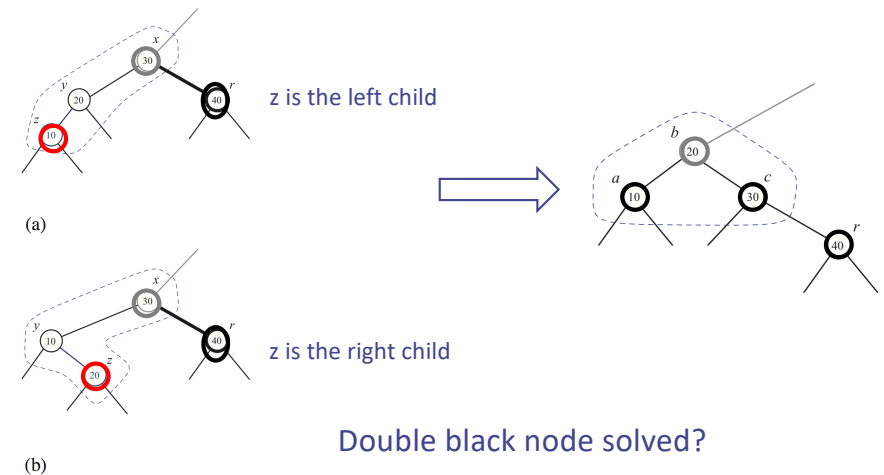
◆ What about deleting a node with a black child?



27

Handling Double Black Nodes: Case 1

- ◆ Case 1: The sibling y of r is black, and has a red child z
- We perform a **restructuring**, and we are done

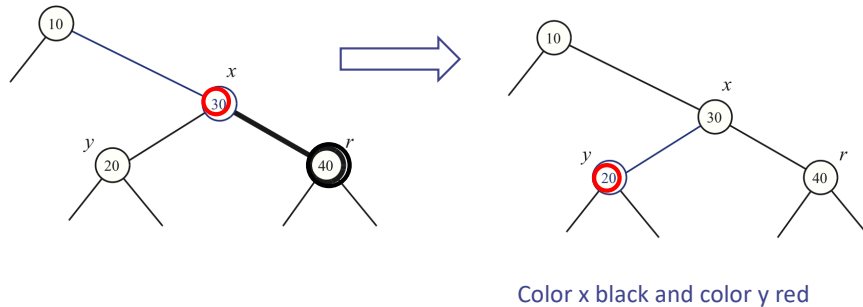


28

Handling Double Black Nodes: Case 2

- ◆ Case 2: The sibling y of r is black, and y 's both children are black

- We perform a **recoloring**
- Case 2-1: x (r 's parent) is red

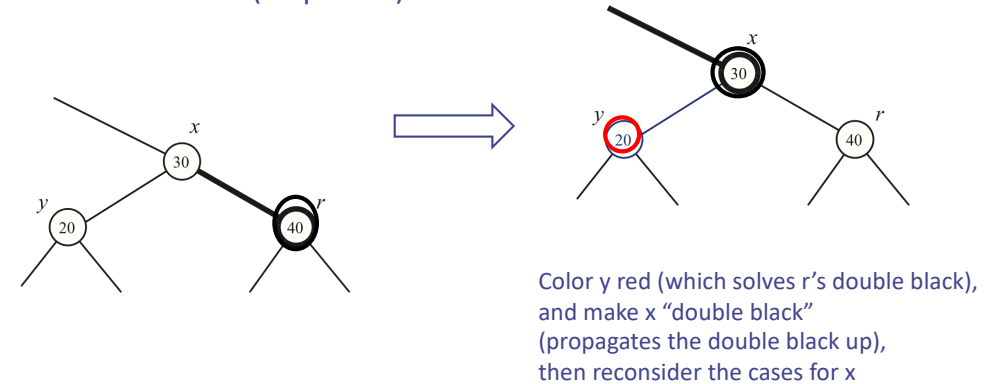


29

Handling Double Black Nodes: Case 2

- ◆ Case 2: The sibling y of r is black, and y 's both children are black

- We perform a **recoloring**
- Case 2-2: x (r 's parent) is black

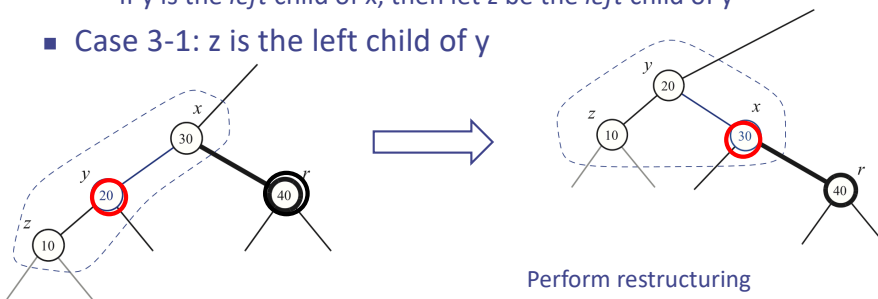


30

Handling Double Black Nodes: Case 3

- ◆ Case 3: The sibling y of r is red

- We perform **adjustment**
 - ◆ If y is the *right* child of x , then let z be the *right* child of y
 - ◆ If y is the *left* child of x , then let z be the *left* child of y
- Case 3-1: z is the left child of y



- Case 3-2: z is the right child of y → Similarly, we apply

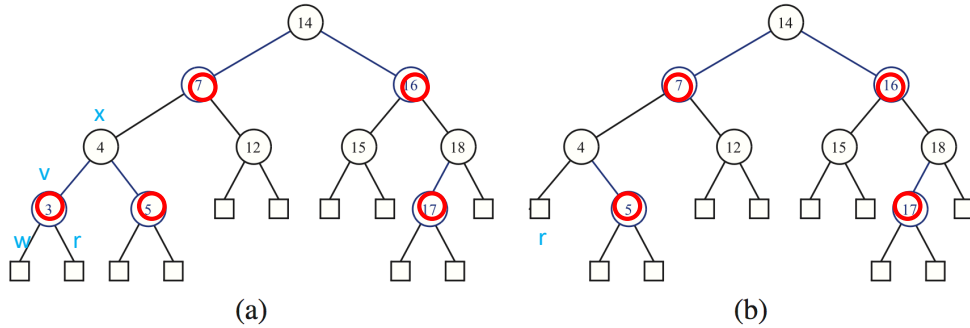
31

Double Black Node Handling: Summary

- ◆ The algorithm for remedying a double black node r with sibling y considers three cases
 - Case 1: y is black and has a red child
 - We perform a **restructuring**, and we are done
 - Case 2: y is black and its children are both black
 - We perform a **recoloring**, 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

32

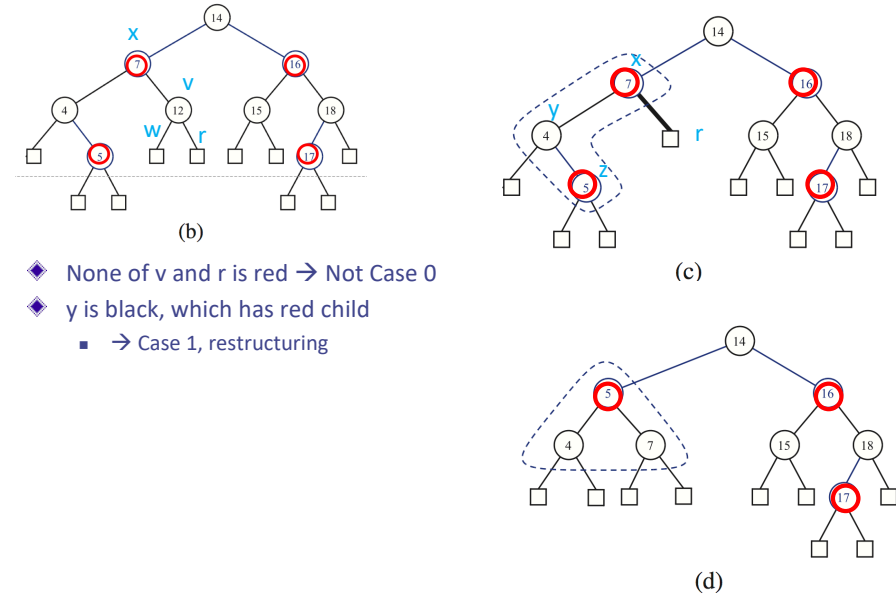
Example: Remove 3



- ◆ v is red \rightarrow Case 0 (either v or r is red)
- ◆ Remove v and w and color r black

33

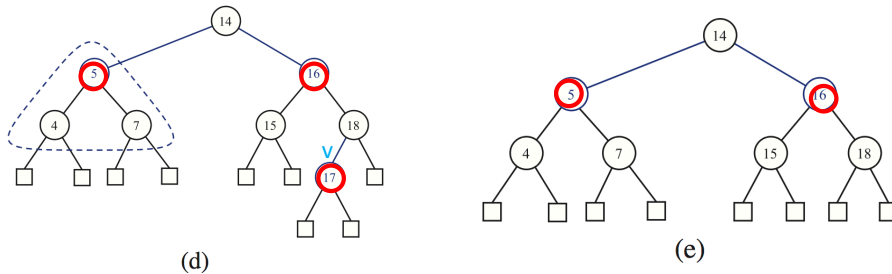
Example: Remove 12



- ◆ None of v and r is red \rightarrow Not Case 0
- ◆ y is black, which has red child
 - \rightarrow Case 1, restructuring

34

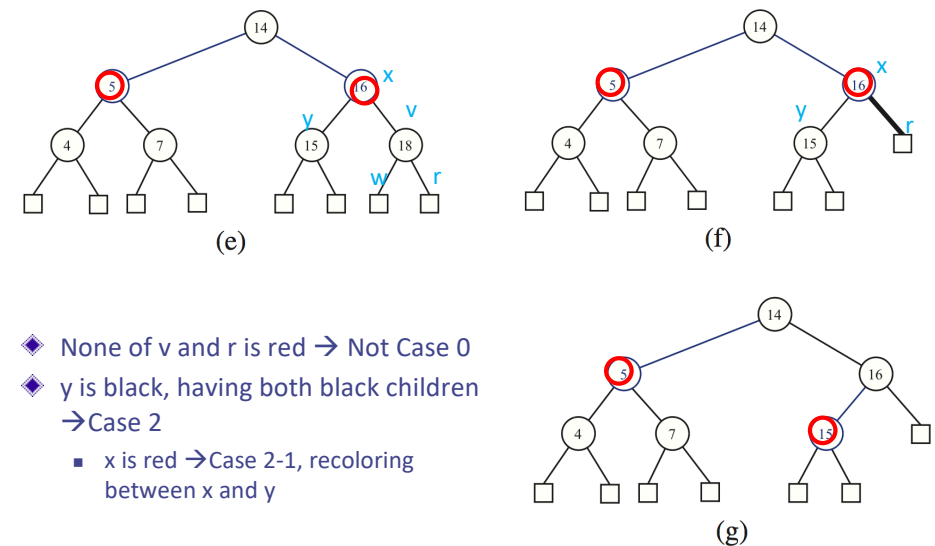
Example: Remove 17



- ◆ v is red \rightarrow Case 0

35

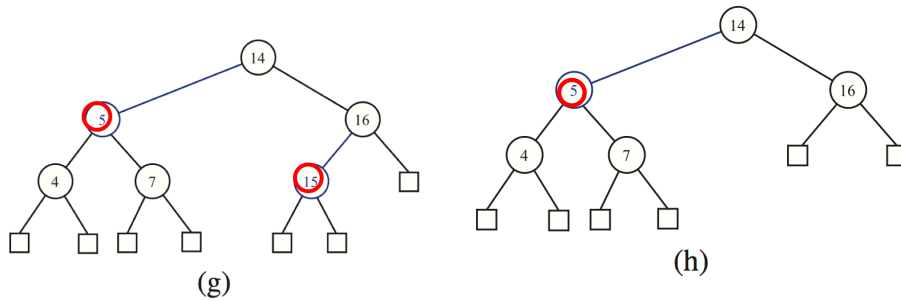
Example: Remove 18



- ◆ None of v and r is red \rightarrow Not Case 0
- ◆ y is black, having both black children \rightarrow Case 2
 - x is red \rightarrow Case 2-1, recoloring between x and y

36

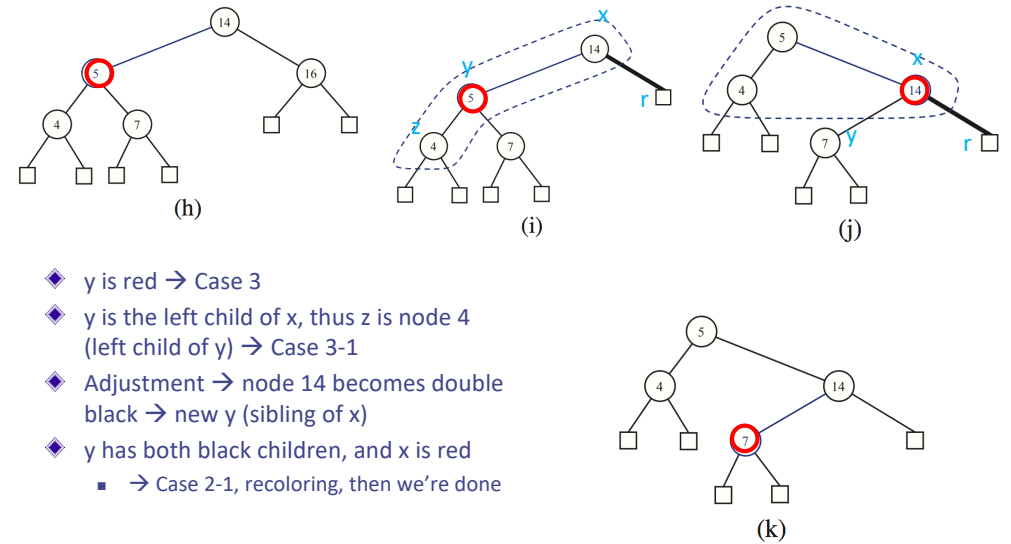
Example: Remove 15



◆ Case 0 (now you know, right?)

37

Example: Remove 16



- ◆ y is red → Case 3
- ◆ y is the left child of x, thus z is node 4 (left child of y) → Case 3-1
- ◆ Adjustment → node 14 becomes double black → new y (sibling of x)
- ◆ y has both black children, and x is red
 - → Case 2-1, recoloring, then we're done

38

Questions?