Lecture 7

- Red-black trees
- Red-black tree properties
- Insert in red-black trees: rotations and recolorings

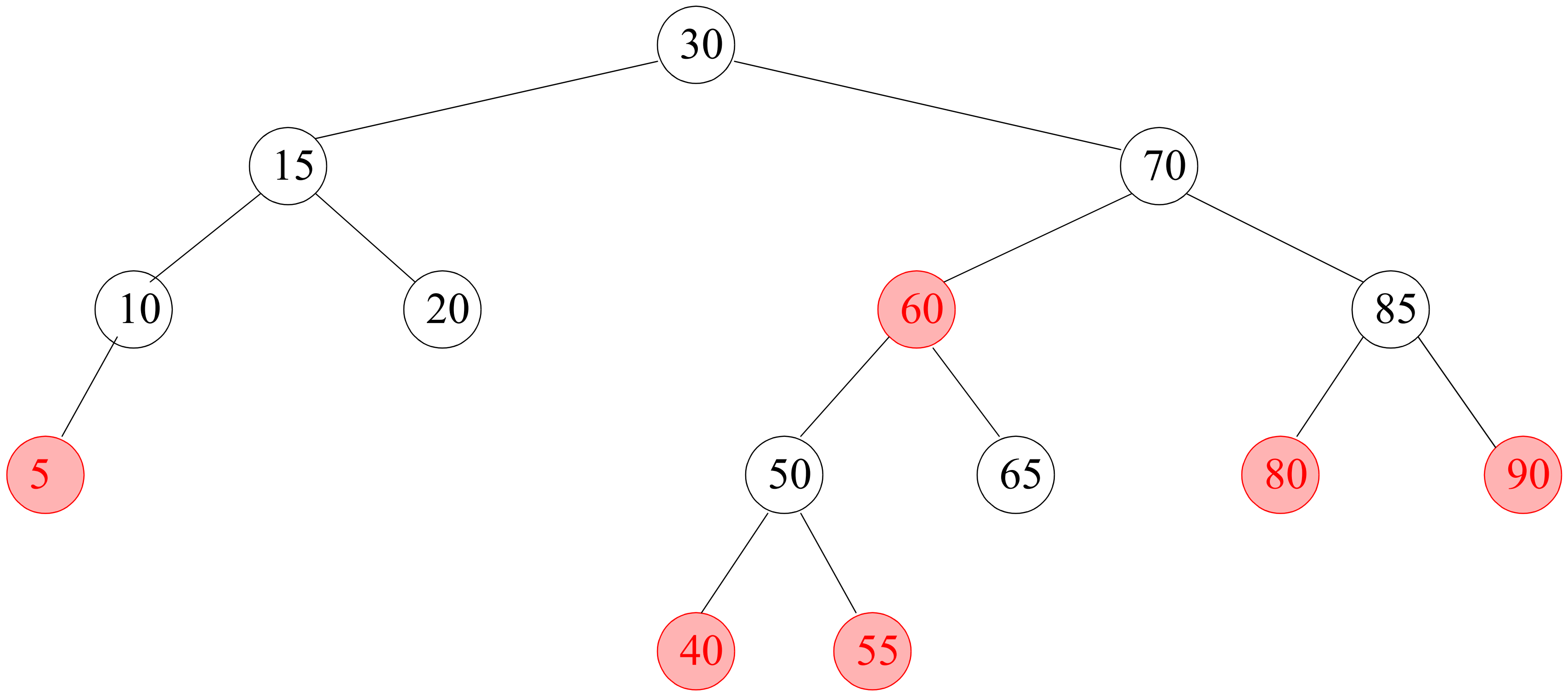
Readings: Weiss, Ch. 12 section 2

Red-Black trees

- A balanced binary search tree approach introduced by Guibas and Sedgewick [1978]
- Red-black trees are high-performance binary search trees that guarantee O(logN) worst-case time cost for insert, find, and delete operations
    - ...the "gold standard" for balanced search trees, used in Java Collections and C++ STL
- A red-black tree maintains these invariants:
    - (1) Every node in the tree is "colored" either *red* or *black*
    - (2) The root of the tree is always black
    - (3) If a node is red, all its children must be black (so can't have 2 consecutive red nodes on any path from the root down to any node)
    - (4) For every node X, every path from X to a null reference (i.e., an empty left or right child) must contain the same number of black nodes

A red-black tree

- Shaded nodes are “red”, unshaded nodes are “black”



- Questions:
  Is this also an AVL tree? Are all red-black trees AVL?
  If this BST were built using the simple BST insert algorithm, which key was inserted first? If it were built using a “balancing” algorithm, can you tell which key was inserted first?

Properties of red-black trees

- The red-black invariants are more complicated than the AVL balance property; however they can be implemented to provide somewhat faster operations on the tree

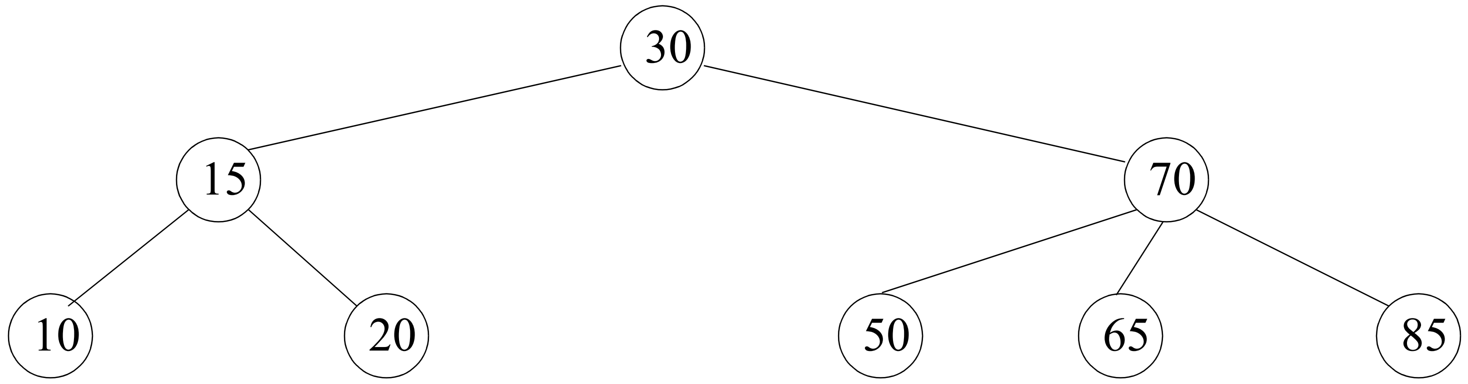
- The red-black invariants imply that the tree is balanced
  - Sketch of proof: eliminate all the red nodes and you have a 4-ary tree that is balanced (every internal node has at least two children, and all leaves are at the same level because of property 4), and adding back the red nodes increases path lengths by at most approximately a factor of two (because of property 3). See next slide

- As a result of the red-black invariants:
  - Red-black trees are balanced; insert, delete, and find operations are O(logN) worst-case. In fact, the height can never be more than $2 (\log_2 N) + 1$
  - It is fairly easy to implement insert and delete operations to be faster by a constant factor than for AVL trees, and to use only one bit of balance information per node
  - The average level of a node in a random red-black tree is comparable to that in a random AVL tree (about $\log_2 N$), so find operations are as fast in the average case

- The trick is to implement Insert and Delete operations that ensure that all the red-black invariants are... invariant

Red-black invariants imply balance

- Sketch of proof:
- Start with a red-black tree with N nodes (example on p. 3)
- Remove all the red nodes, moving children of a red node up to be children of its parent:



- Result: a tree containing at most N nodes, with all internal nodes having at least 2 children, and all leaves at the same level, so its height is at most $\log_2 N$
- Now put the red nodes back: this can increase the height by at most a factor of 2, plus 1
- So, a red-black tree with N nodes has height at most $2 (\log_2 N) + 1$

Insert in red-black trees

- It is possible to implement Insert in red-black trees ‘top down’ so that all needed operations are done while descending the tree from the root down to where the new node will be inserted
    - This permits an efficient, iterative implementation of Insert
- Contrast with Insert in AVL trees: Descent from the root finds the place to insert the new node; then detecting failures of the AVL property and fixing them is done on the way back up to the root
    - This means in effect travelling the path from root to leaf twice, whether it is done recursively, or iteratively (by using parent pointers)
- As a result red-black insertion is typically somewhat faster than AVL insertion
- Red-black insertion uses single and double AVL rotations, plus “color-change” operations, to ensure the red-black properties are invariant

Insertions in red-black trees: color of leaf node

- A new red-black tree node will be initially inserted as a leaf node, using the usual binary search tree insert algorithm
- What color should the new leaf node be?
  - Well, if it’s the root of the tree (first insertion in an empty tree), it must be black
  - But what if it is a non-root leaf node?
  - If we make it black, then we will violate property (4), the number-of-black-nodes-on-a-path-to-null property: we will have created a path to a null reference (the new leaf node’s null child pointers) that contains one more black node (the new leaf itself) than another already existing path
- So, it appears that a new non-root leaf node should be red

Insertions in red-black trees: color of parent of leaf node

- Okay, so a newly inserted (non-root) leaf node should be red
- What if the parent of the new leaf node is black? Example: insert 25 in the example tree shown
- Then you are done! (You should be so lucky.) All the red-black properties are preserved.
- But what if the parent of the new leaf node is red?
- Then the insertion has violated property (3). Let’s look at how to fix it

Parent of new leaf is red, sibling of parent is black

- We are inserting a new leaf node, which will be colored red, in a red-black tree
- The parent of this new leaf node is also red; this violates the red-black invariants, so it is a problem which we must fix
- We will work with the assumption that *parent's sibling is black*
- A little later, we will worry about how to ensure that this assumption holds
- (Note: in considering cases, we will take null references (empty children) to be equivalent to nodes colored black. So for example the assumption that the parent's sibling is black is an assumption that either the parent's sibling is black, or the parent has no sibling.)

Fixing up the neighborhood of a leaf

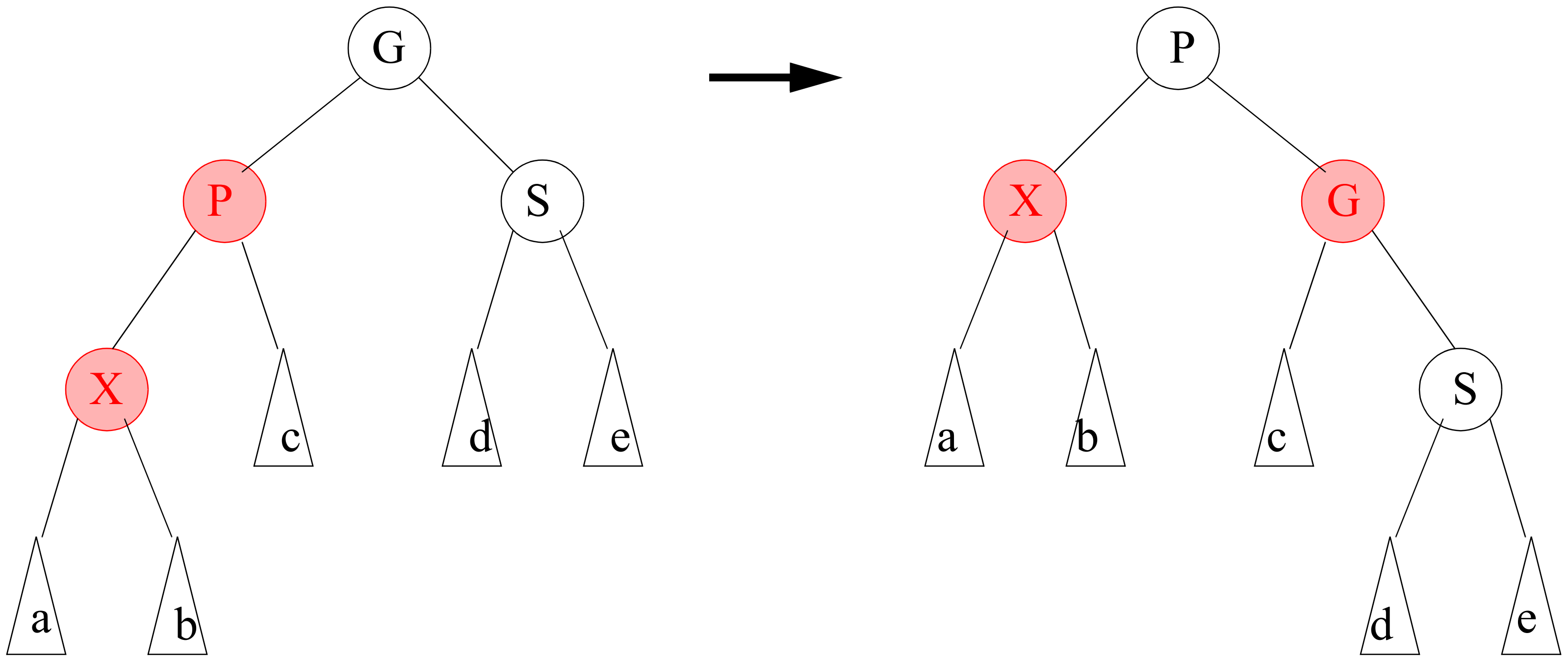
- Terminology:
  - the new node is X, which we are initially coloring red
  - the parent of X is P , which is red
  - the sibling of P is S , which is black
  - the parent of S and P (the grandparent of X) is G , which must be black

- There are 4 cases to consider:
  - Case 1: P is left child of G, and X is left child of P (X is “outside grandchild” of G)
  - Case 2: P is left child of G, and X is right child of P (X is “inside grandchild” of G)
  - Case 3: P is right child of G, and X is left child of P (X is “inside grandchild” of G)
  - Case 4: P is right child of G, and X is right child of P (X is “outside grandchild” of G)

- We will look at cases 1 and 2. Cases 3 and 4 are symmetric with these

Fixing up the neighborhood of a leaf: case 1

- The Case 1 situation can be fixed with a single right rotation between P and G, and flipping the colors of nodes P and G:



- Note: X is shown with subtrees, and node S is shown, for generality (if X is a new leaf, of course subtrees a and b will be empty, as will subtree c, and S will not exist)
- How can we be sure this single rotation and recoloring fixes all the red-black properties?

Thinking about Case 1

- Before the rotation and recoloring:
  - Node X is red; its parent P was also red (that was the problem)
  - P was red, so its parent G must have been black (because of property 3)
  - S was black, by assumption

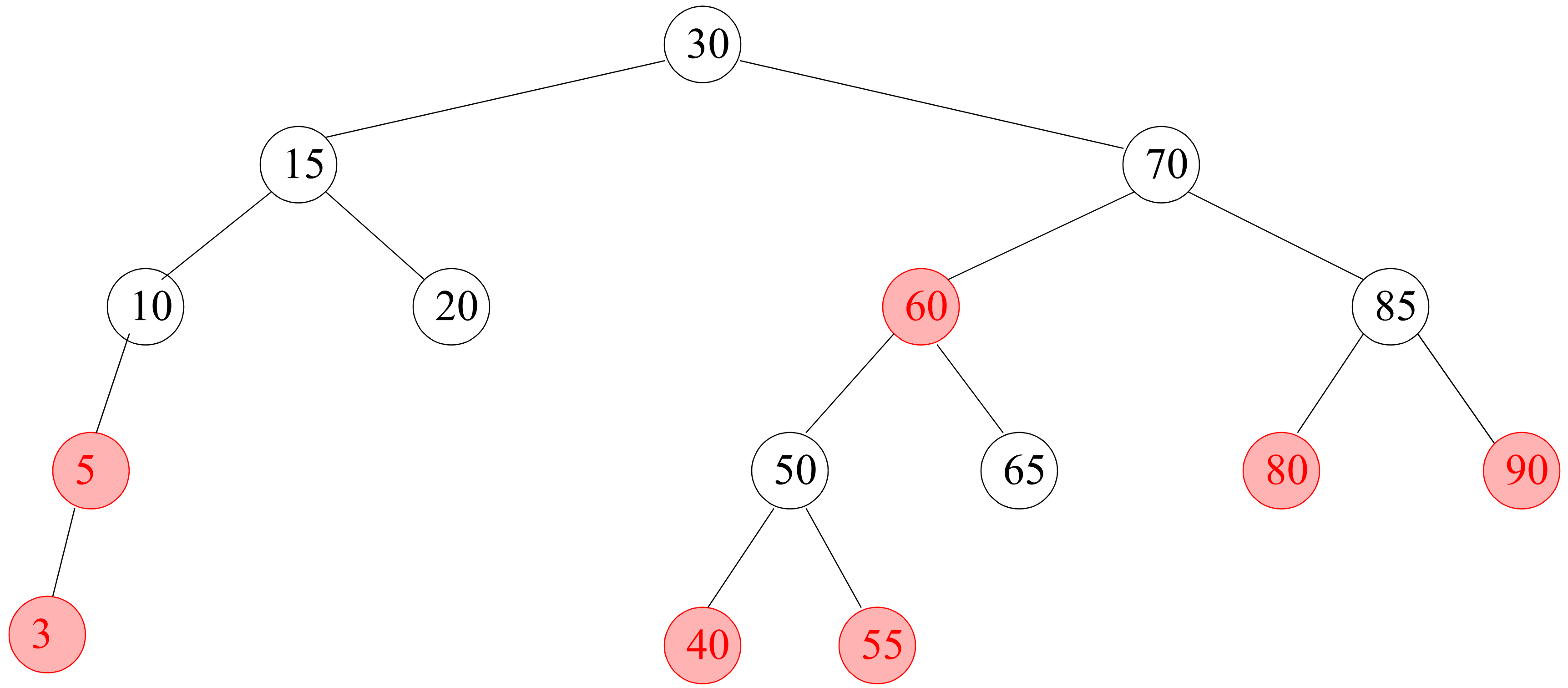
- After the rotation and recoloring:
  - X is still red, but its parent is now black (fixed that problem)
  - G is now red, but its parent is black (that's good)
  - The children of the now-red G are S (which is black by assumption) and the former right child of P (which must be black, since P was formerly red)

- So property 3 has been fixed up. What about property 4?
  - Check: the number of black nodes on every path from the root to subtrees a,b,c,d,e cannot have been changed by this rotation and recoloring

- Does the binary search tree ordering property still hold? Yes, because the rotation is just an AVL rotation

Case 1: an example

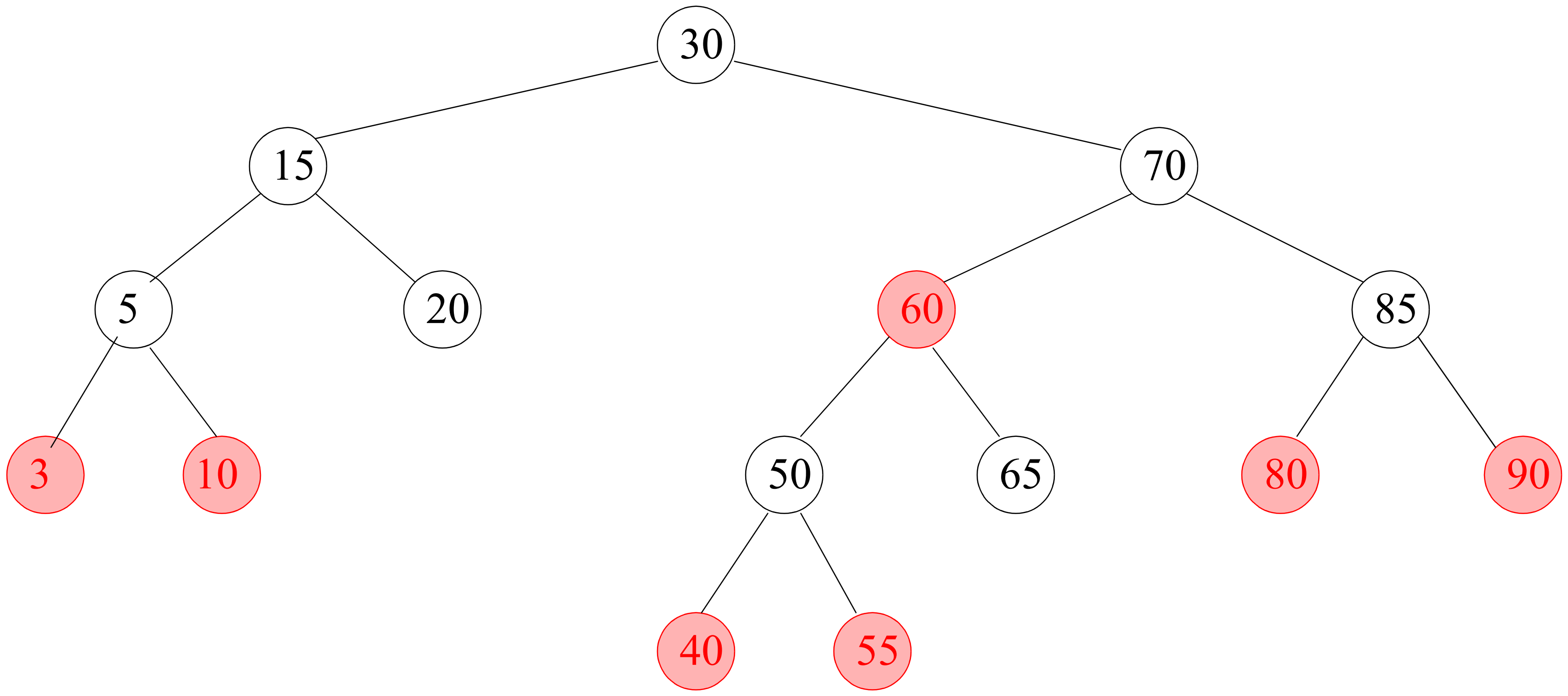
- Insert 3 in the tree:



- Fix it!

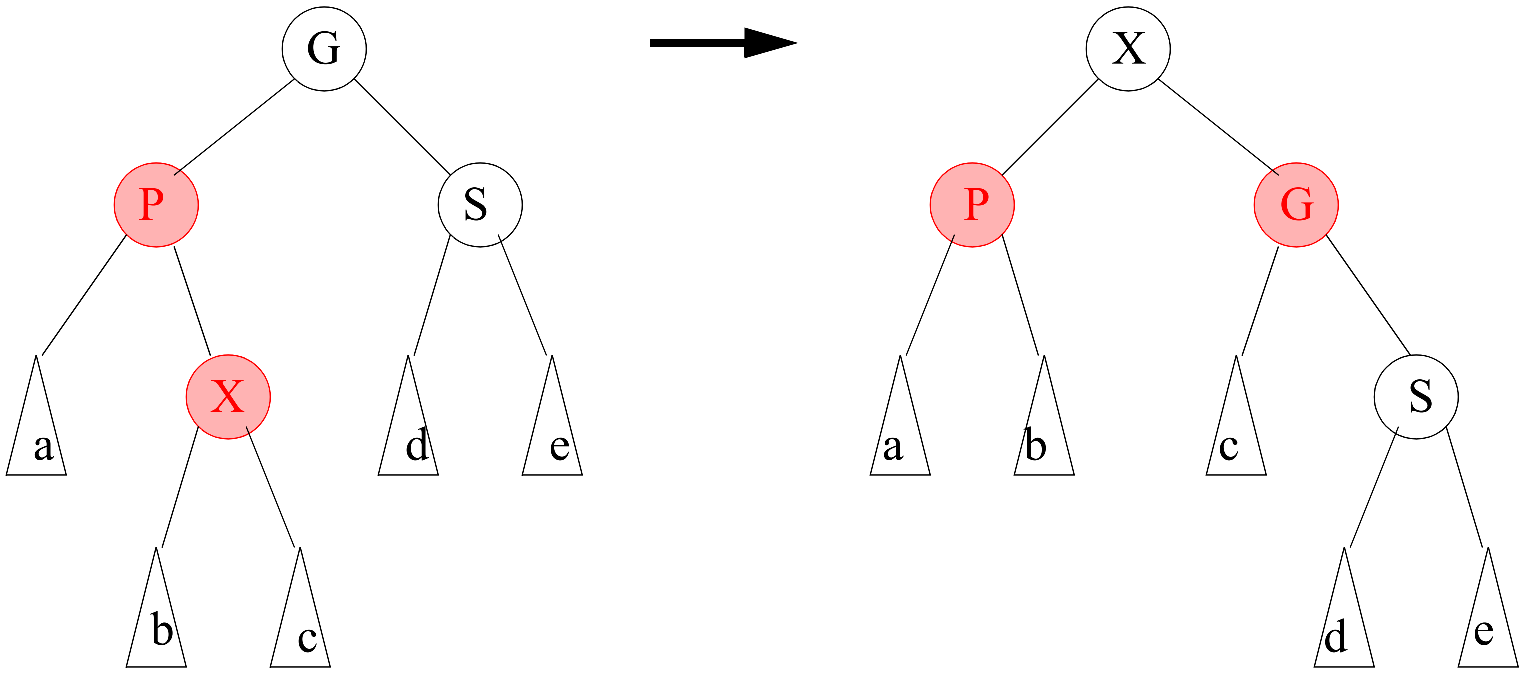
Case 1: an example, cont'd

- The result of the single rotation and recoloring:



Fixing up the neighborhood of a leaf: case 2

- The Case 2 situation can be fixed with a double left rotation between X, P and G, and recoloring of nodes X and G:



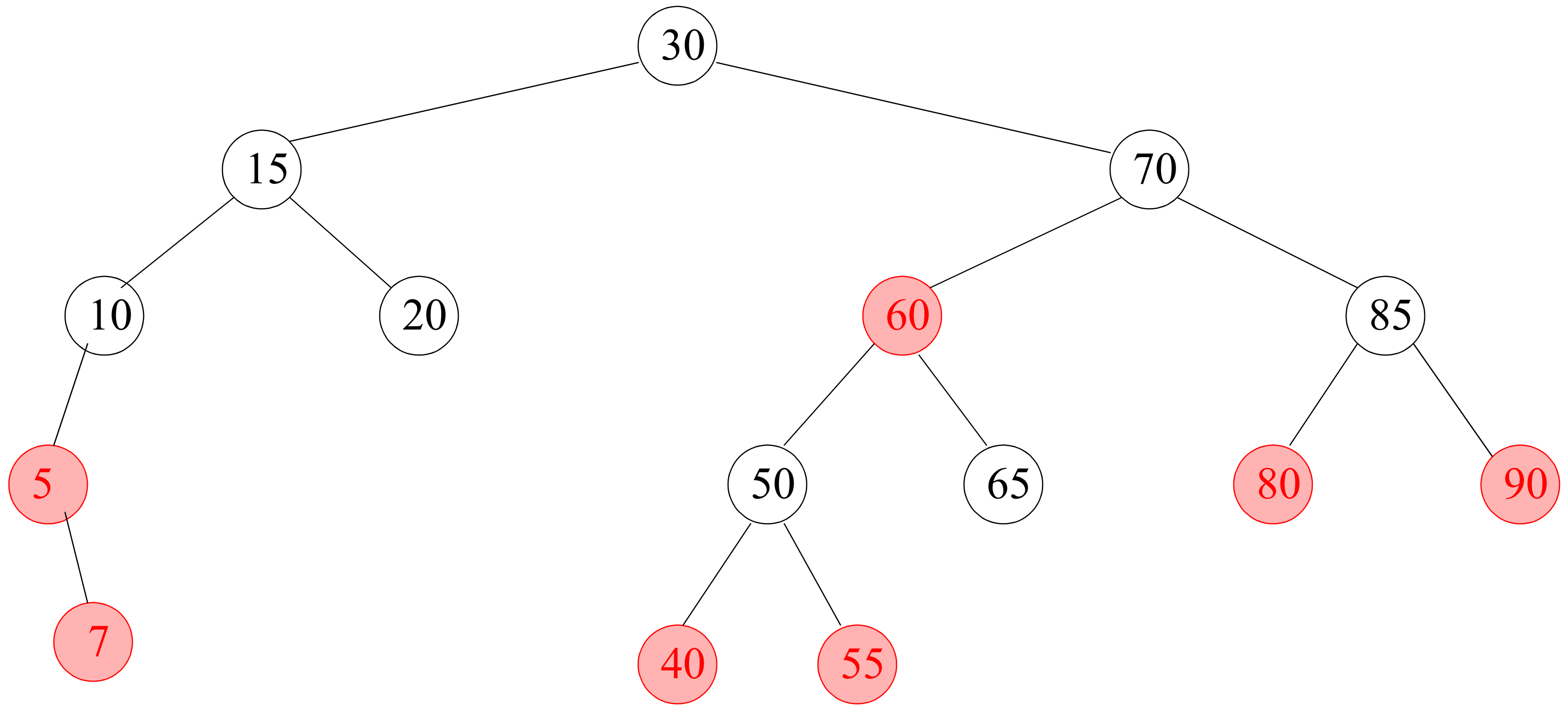
- Note: X is shown with subtrees, and node S is shown, for generality (if X is a new leaf, of course subtrees b and c will be empty, as will subtree a, and S will not exist)
- How can we be sure this double rotation and recoloring preserves/fixes all the red-black properties?

Thinking about Case 2

- Before the rotation and recoloring:
  - Node X is red; its parent P was also red (that was the problem)
  - P was red, so its parent G must have been black (because of property 3)
  - S was black, by assumption
- After the rotation and recoloring:
  - X is now black
  - G is now red, but its parent is black (that's good)
  - The children of the now-red G are S (which is black by assumption) and the former right child of X (which must be black, since X was formerly red)
- So property 3 has been fixed up. What about property 4?
  - Convince yourself that the number of black nodes on every path from the root to subtrees a,b,c,d,e cannot have been changed by this rotation and recoloring
- Does the binary search tree ordering property still hold? Yes, because the double rotation used is just two single AVL rotations

Case 2: an example

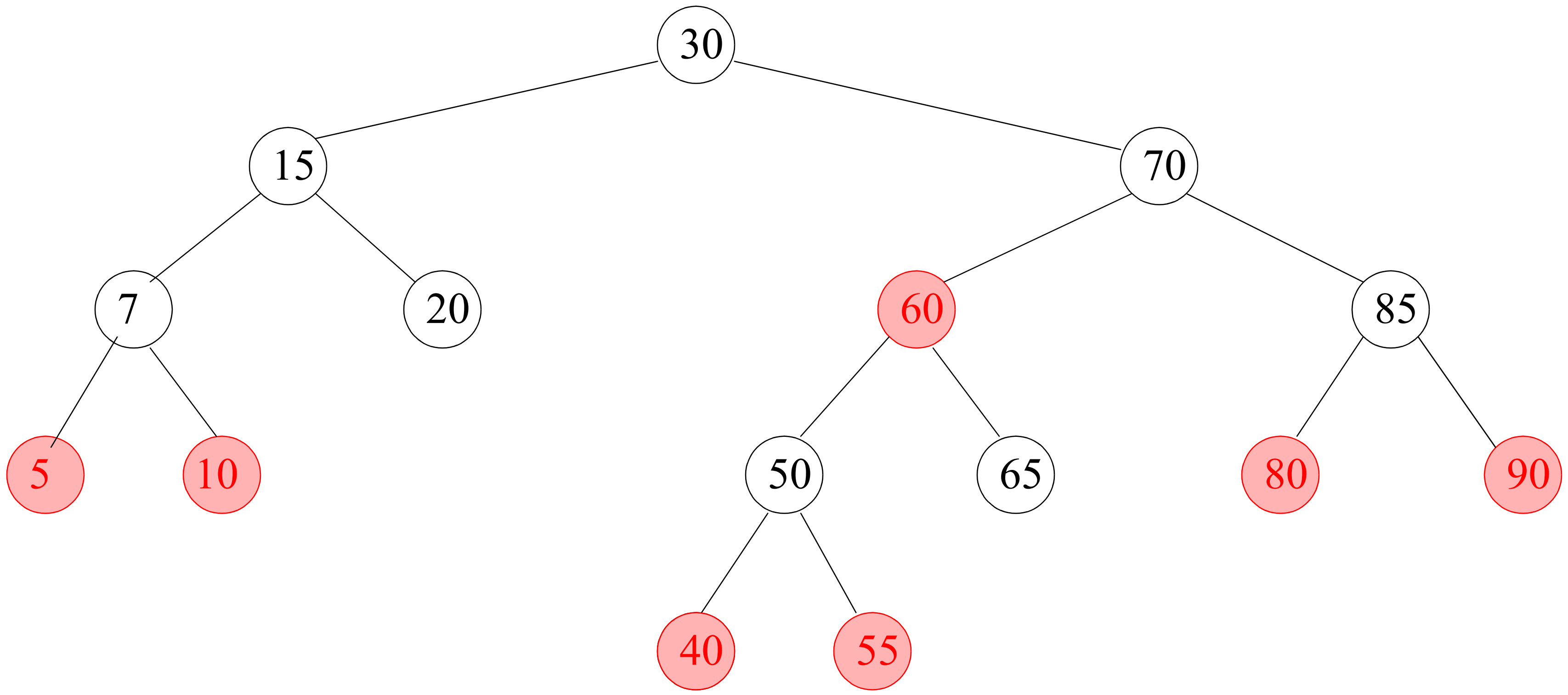
- Insert 7 in the tree:



- Fix it!

Case 2: an example, cont'd

- The result of the double rotation and recoloring:



Making sure sibling S is black

- The operations we've just looked at will fix things up in the neighborhood of a new leaf node... but to work they require that S, the sibling of the new node's parent, is black
- If S is red, these operations may not restore all the red-black invariants
- The trick: As we descend the tree looking for the place to insert the new node, we will make changes if we find a node on the path that has 2 red children
- We will do this in a way that guarantees that when we get to the place where we are inserting the new red leaf node X, we can be sure that the grandparent of X does not have 2 red children, and so either:
  - X's parent is black, and there is no problem to fix; or,
  - X's parent's sibling S is black, and the Case 1-4 rotations/recolorings will work
- So, how to ensure that?
- Actually, the Case 1-4 rotations/recolorings suffice here too!

The red-black insertion algorithm

- So, the red-black algorithm for insertion is this:

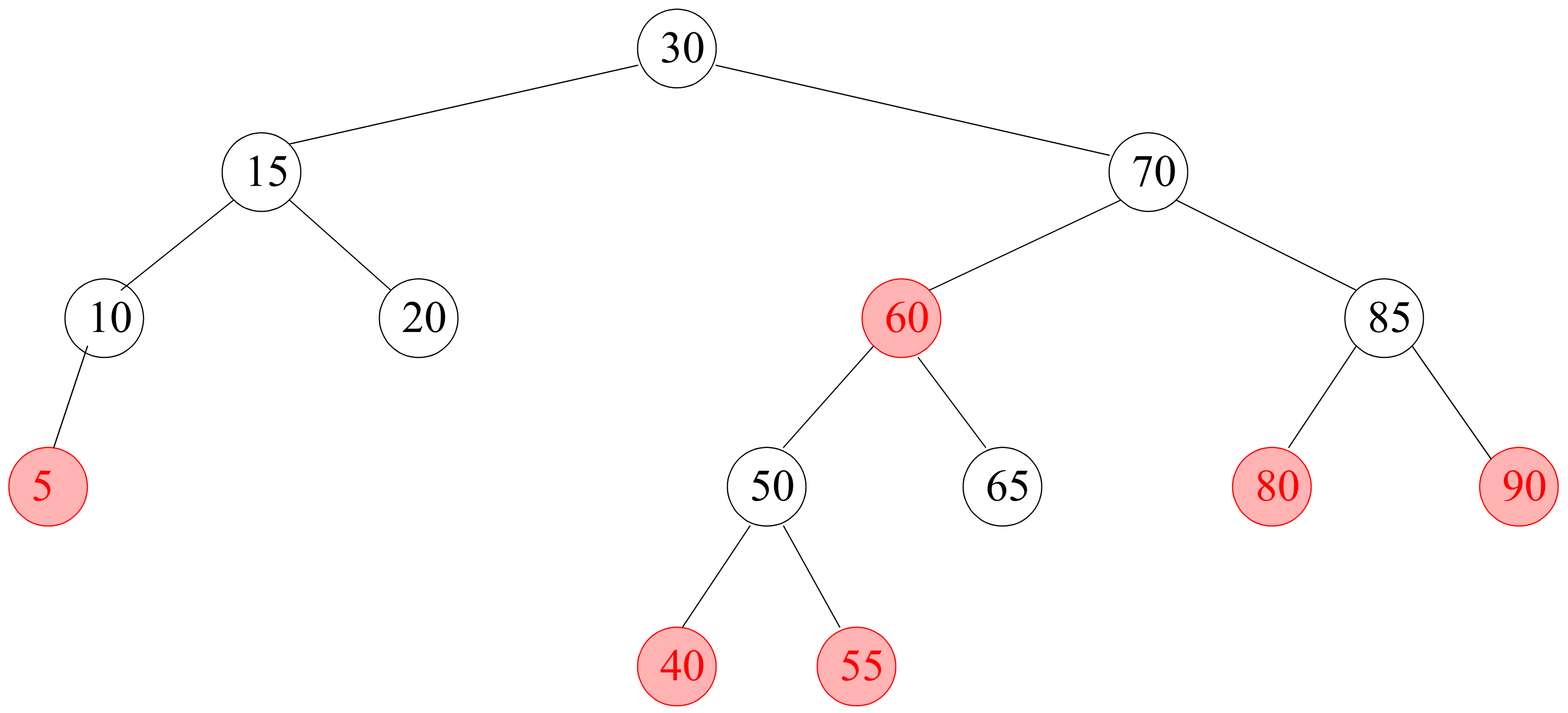
- Descend the tree from the root

- At each node X, if X has two red children (X must then be black), do the following:
  - change X to red, and its children to black
  - if X's parent is black, continue; all is okay so far (you should understand why!)
  - else, consider X's parent P, parent's sibling S, and grandparent G as one of Case 1-4, and perform the appropriate rotation and recoloring (this will work, because S must be black; both P and S can't be red, or we would have changed them at a higher level)
  - now continue descending from X

- When you get to the null reference where the new node will be inserted, create a new red node, insert it there, and apply Case 1-4 to the new node as needed. Done!

Inserting in a red-black tree

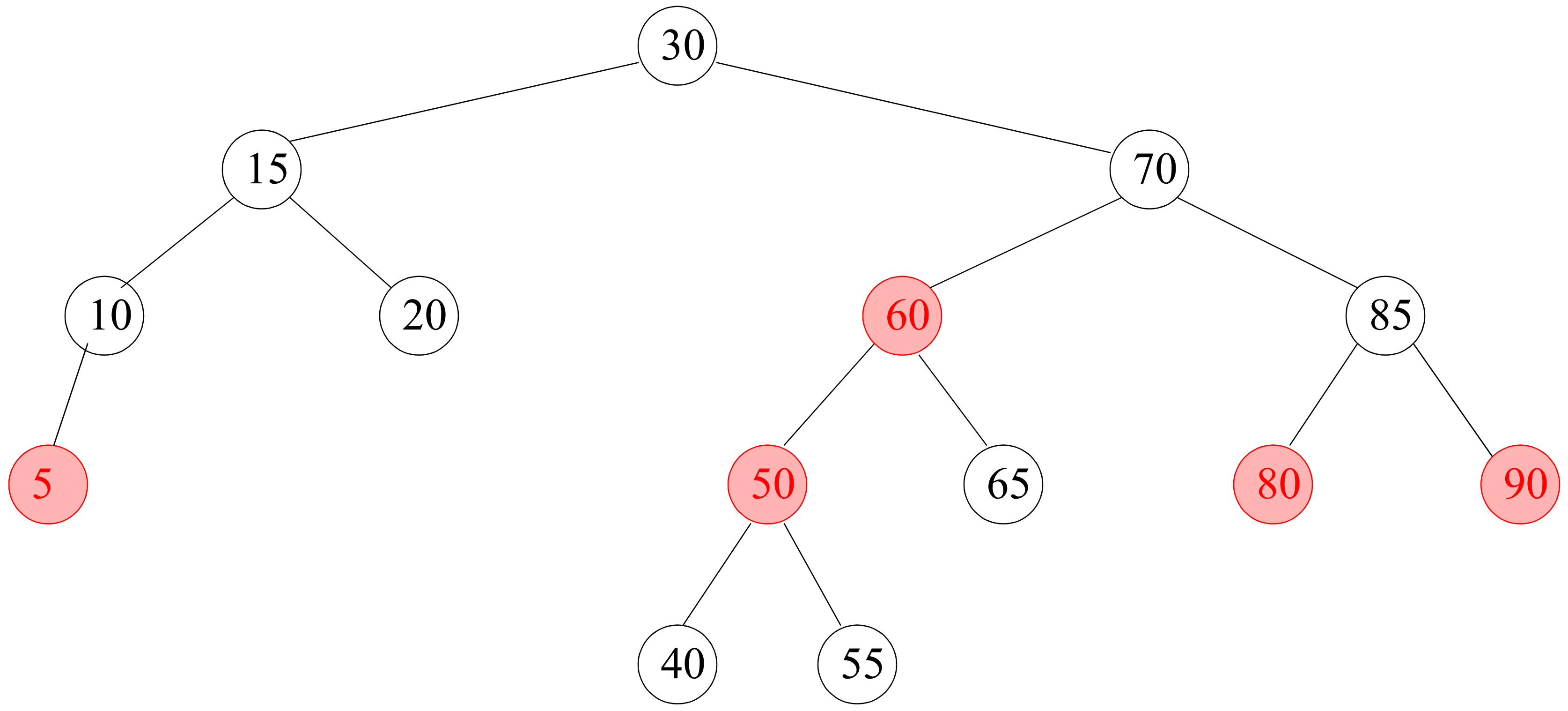
- Insert 45 in this red-black tree:



- Descend from the root: 30, 70, 60, 50. 50 has two red children... We need to change this before proceeding.

Inserting in a red-black tree, cont'd

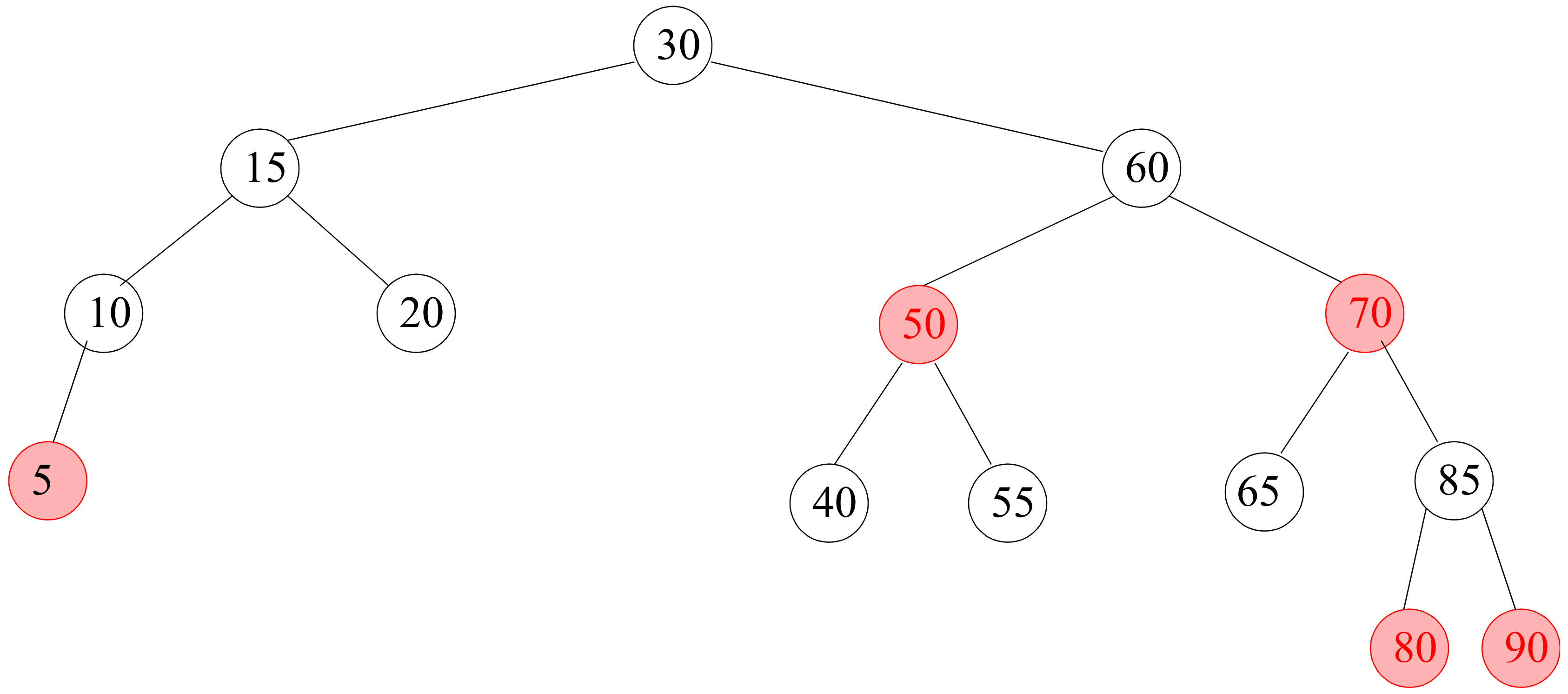
- Change 50 to red,and its children to black:



- Now 50 is red, and has a red parent (with of course a black sibling). Case 1 applies!

Inserting in a red-black tree, cont’d

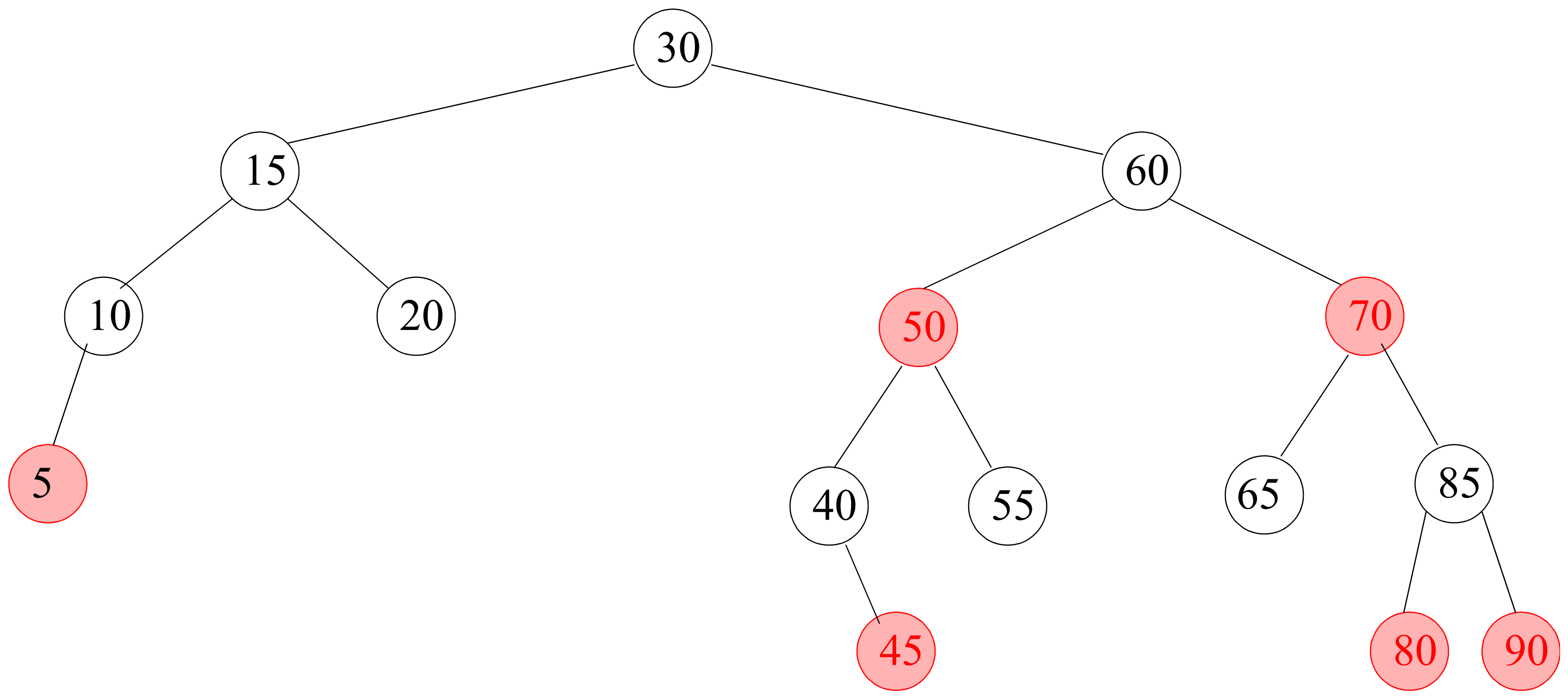
- Do a single right rotation, and recolor:



- Now continue descending the tree, starting where we left off: node containing 50

Inserting in a red-black tree, the end

- The key 45 is inserted in a new red right child of 40:



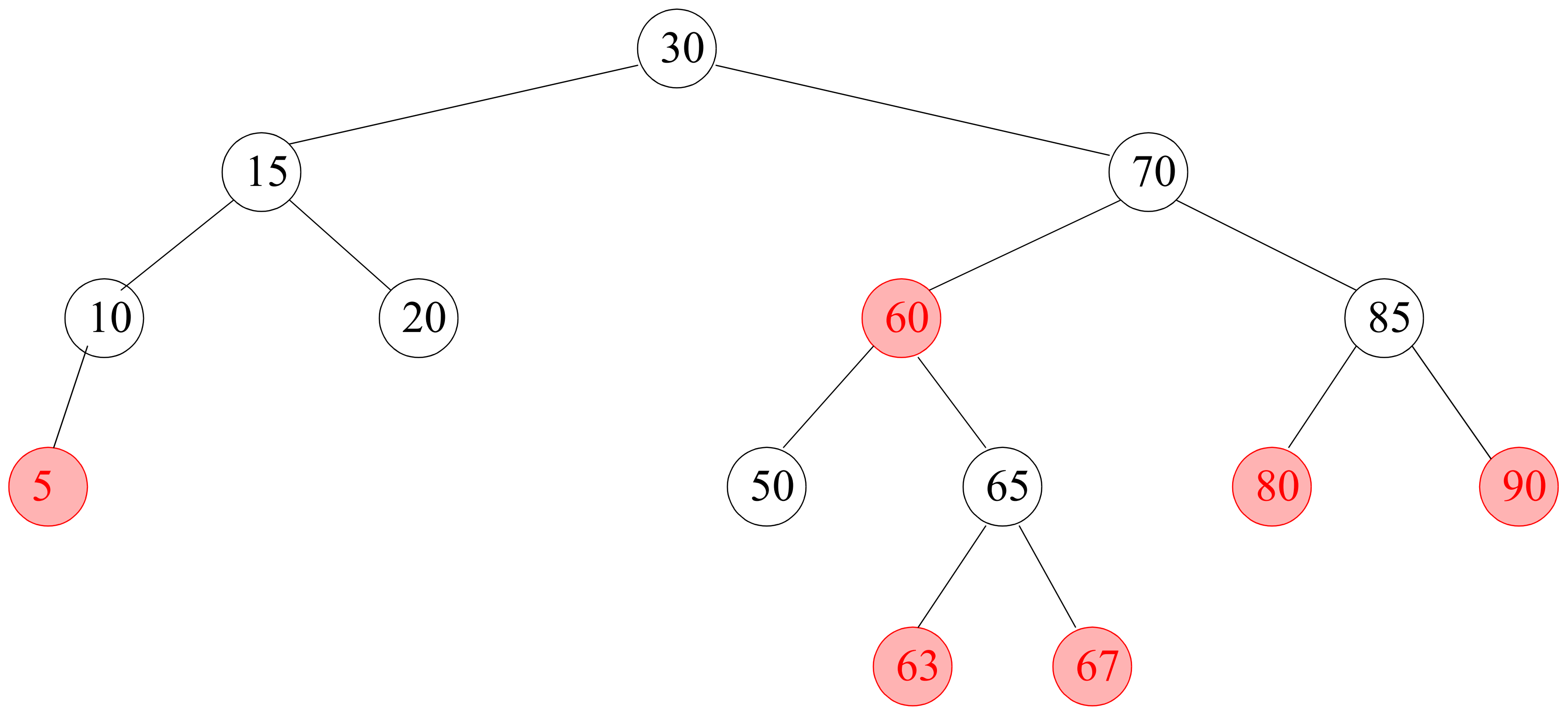
- In this example, since the new leaf's parent is black, there is nothing more to do (if not, we would have to apply one of Case 1-4). Done!

No node with 2 red children?

- As we decend the tree, we detect if a node X has 2 red children, and if so we do an operation to change the situation
- Note that in doing so:
  - we may change things so that a node above X now has 2 red children, where it didn't before! (example: node 60 on previous page)
  - if we have to do a double rotation, we will move X up and recolor it so that it becomes black, and has 2 red children itself! (example: work through inserting 64 in the tree on the following page)
- But neither of these is a problem, because
  - it never violates any of the properties of red-black trees (those 2 red nodes will always have a black parent, for example),
  - and the 2 red siblings will be too "high" in the tree for either of them to be the sibling of the parent of any red node that we find or create when we continue this descent of the tree

Inserting in a red-black tree, another example

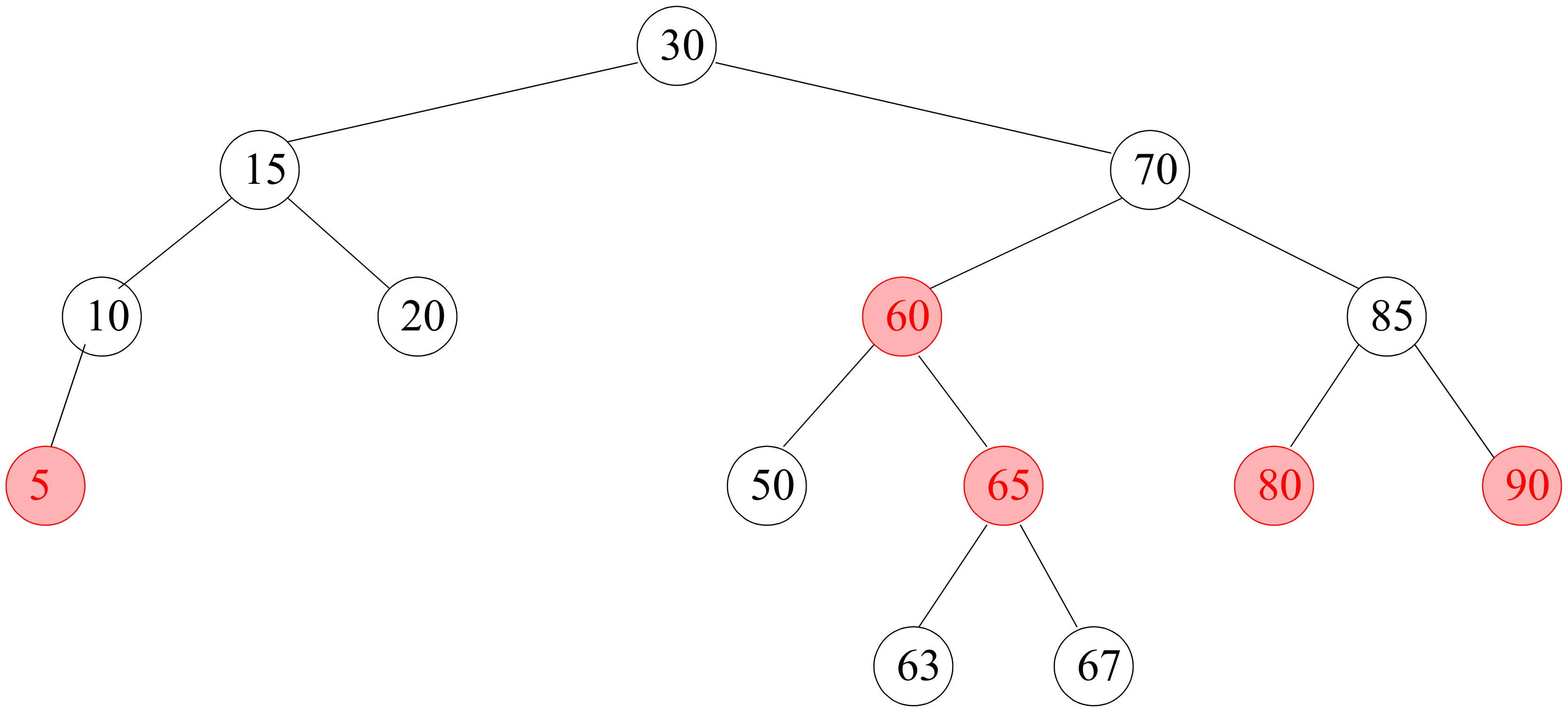
- Insert 64 in this red-black tree:



- Descend to node containing 65; note it has 2 red children; case 2 (X is an inside left grandchild) applies

Inserting in a red-black tree, another example, frame 1

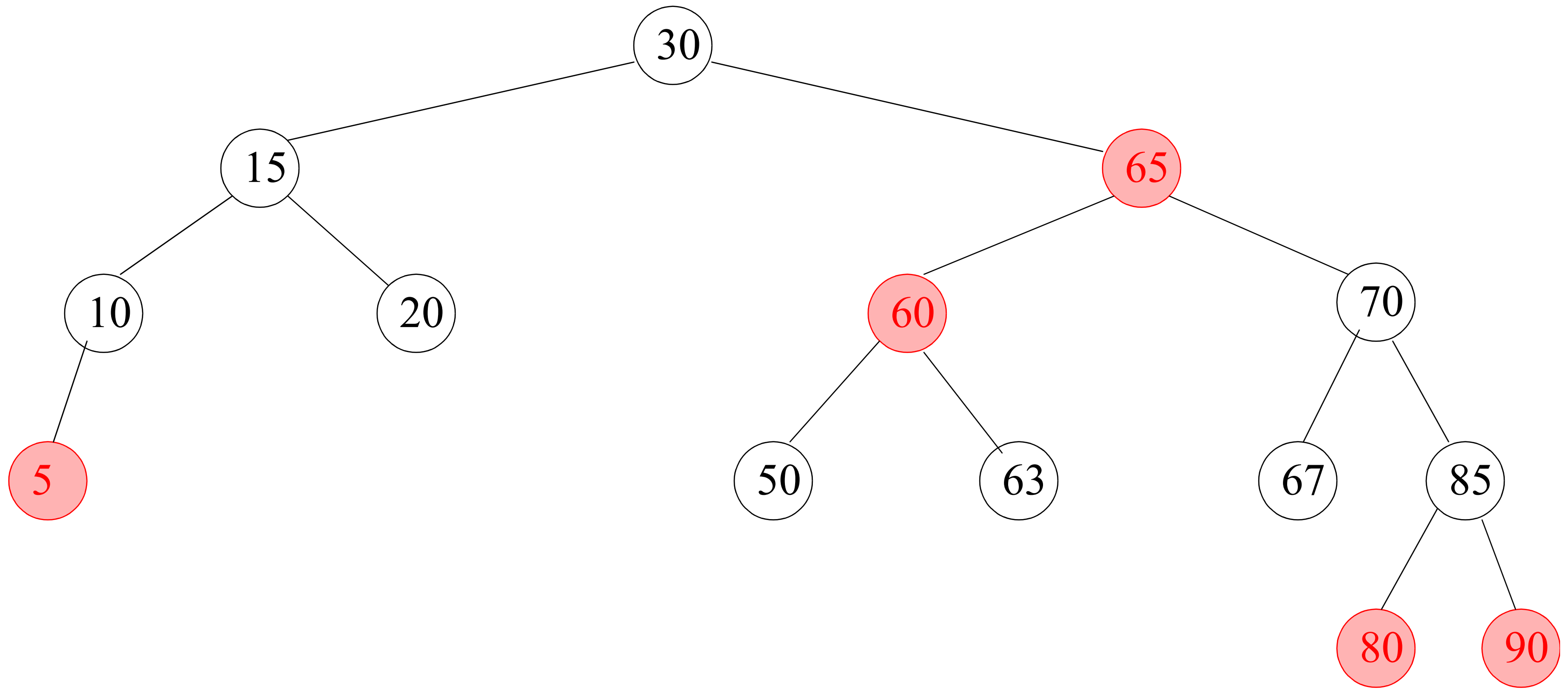
- Recolor node X (the node containing 65) and its children



- X's parent is now also red, so we do a double rotation of X with its parent and grandparent, and recolor

Inserting in a red-black tree, another example, frame 2

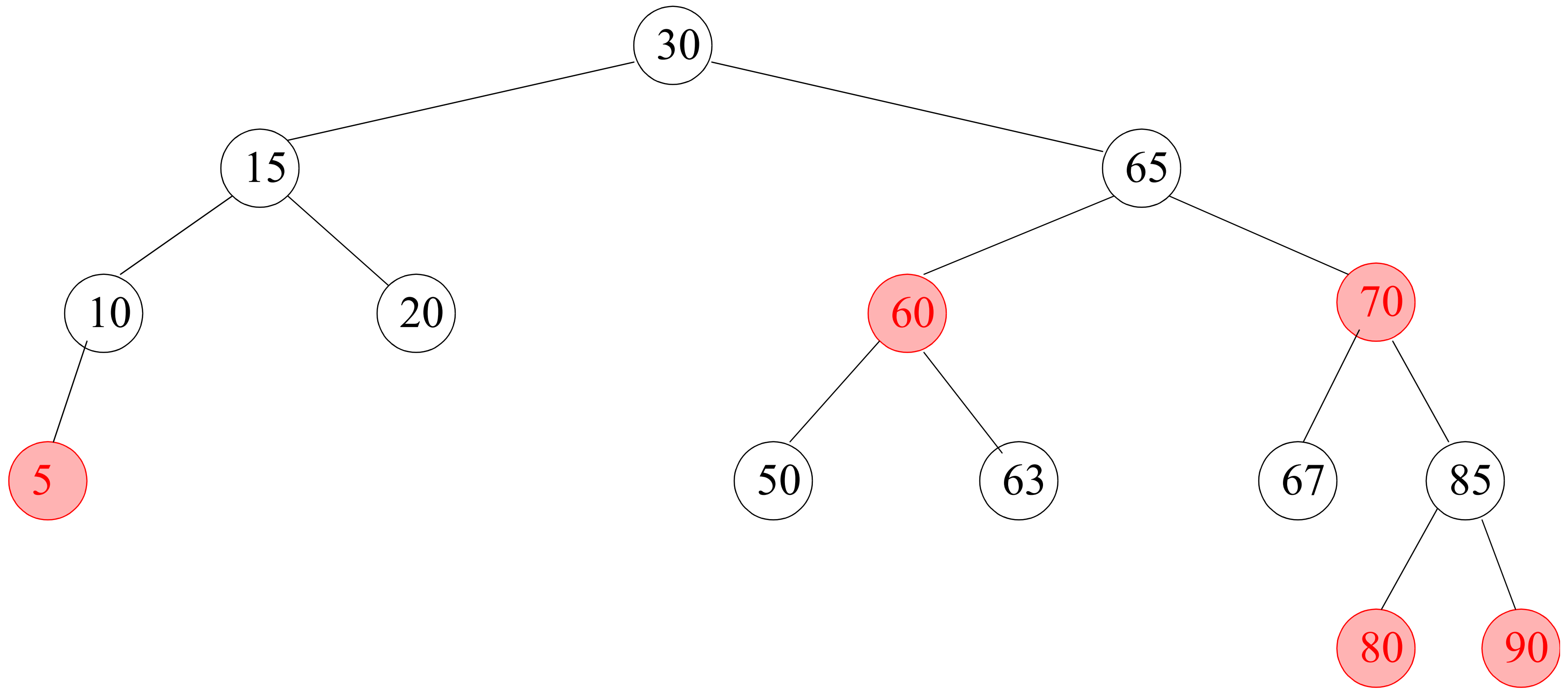
- After double rotation:



- ... still need to recolor X and G

Inserting in a red-black tree, another example, frame 3

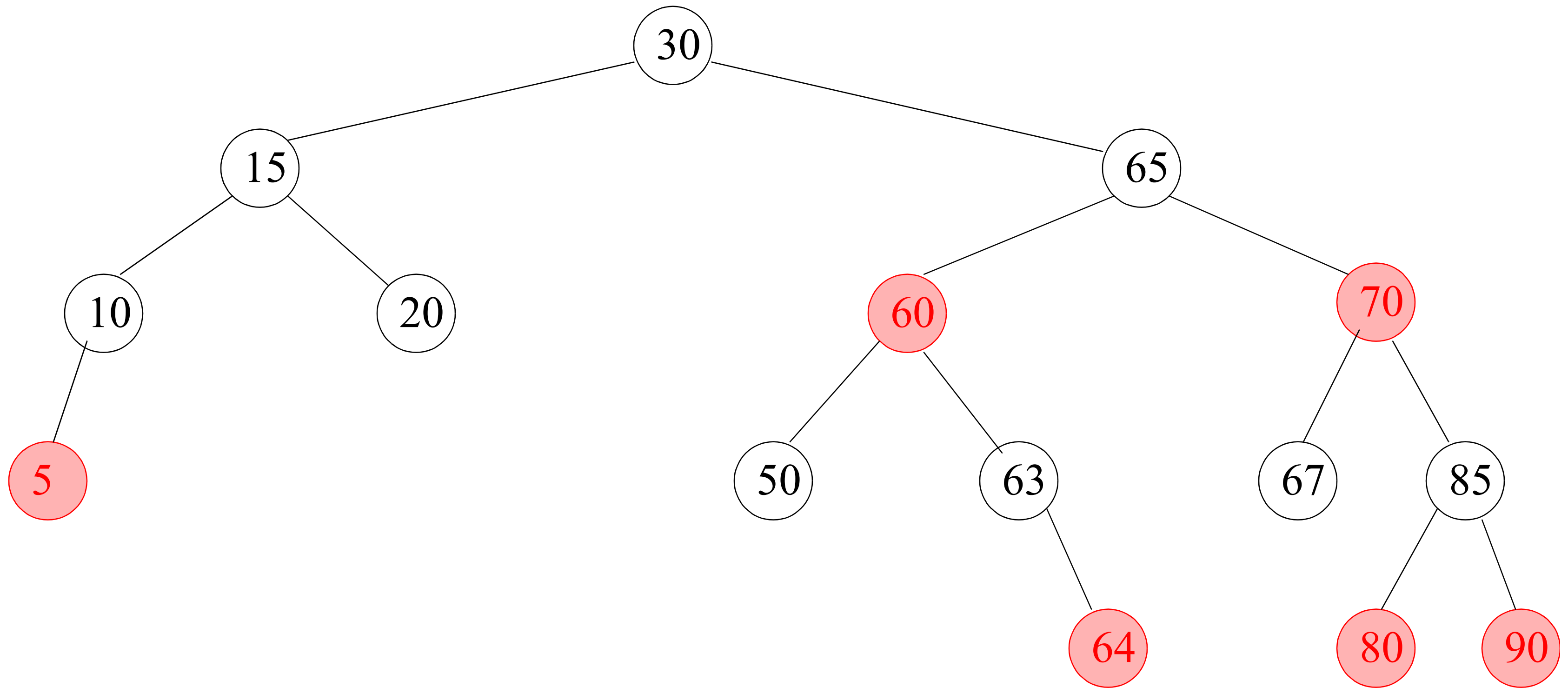
- After recoloring... node “X” -- the one containing key 65 -- still has 2 red children!



- But we can just continue the descent because X must now have 4 grandchildren, all black; so neither of these 2 red siblings will be parent of the new leaf when we insert it

Inserting in a red-black tree, another example, frame 4

- Descend and insert 64 as a new red leaf



- ...done.

Implementing red-black tree operations

- In implementing red-black tree operations, we have seen that there are different cases that need to be considered
- There are even more special cases you may need to consider: if you are inserting a node in an empty tree, or if you are doing an insertion that is a child of the root (that new node has no grandparent!), etc.
- Some of these special cases can be handled more easily (they are not special anymore) if you use "sentinel" nodes in your implementation
  - a sentinel root node with key +INFINITY, so all "real" nodes are in its left subtree
  - sentinel null nodes, colored black
- The delete operation is even somewhat more complicated than insert to handle correctly, unless you use "lazy deletion"...

Delete in binary search trees: lazy deletion

- A simple approach is “lazy deletion”:
  - Nodes contain a boolean field indicating if they are deleted or not
  - To delete a key from the tree, just find the node containing that key and mark it as deleted

- Advantages:
  - Makes the delete operation simple to implement

- Disadvantages:
  - Insert, Find, etc. need to be modified to do the right thing with ‘deleted’ nodes that they encounter
  - Most importantly, if there are many delete operations without re-inserting the same keys, lazy deletion is very wasteful of space: deleted nodes are never really removed from the tree
  - This makes lazy deletion wasteful of time as well: deleted nodes remain in the tree and can increase the length of paths in the tree compared to what they would be with those nodes removed

Summary of red-black trees

- In his skip list paper, Pugh says
  - "Implementing balanced trees is an exacting task and as a result balanced tree algorithms are rarely implemented except as part of a programming assignment in a data structures class"
- It is true that it is a lot of work to implement balanced trees.... but once they have been implemented, debugged, and made part of a standard library, they are very nice to have
- ... And red-black tree implementations are now part of the C++ Standard Template Library, and the Java Collections Framework
- If you need guaranteed O(log N) performance, have data from a well-ordered set, and are writing in C++ or Java, these library implementations are very good
- If you can settle for "almost certainly guaranteed" O(logN) performance, have data from a well-ordered set, and can't use the standard libraries in C++ or Java, skip lists or randomized search trees work very well for significantly less programming effort

Next time

- Trees for representation
- Tries, decision and classification trees, discrimination nets
- Huffman coding

Reading: Weiss, Ch 10