ECE430.217 Data Structures

Red-black trees

Byoungyoung Lee

https://compsec.snu.ac.kr

byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

In this topic, we will cover:

- The idea behind a red-black tree
- Defining balance
- Insertions and deletions
- The benefits of red-black trees over AVL trees

Red-Black Trees

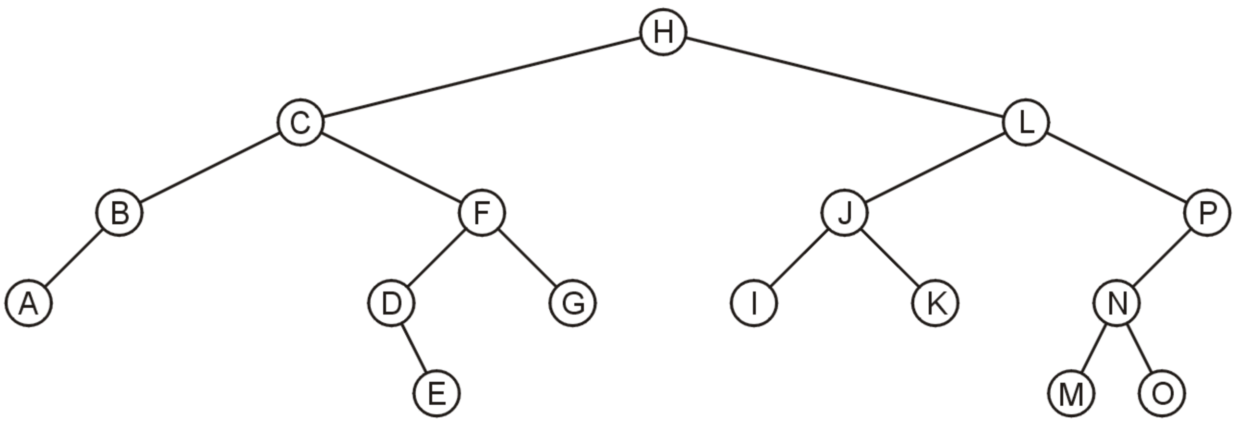
A red black tree “colours” each node within a tree either red or black

- This can be represented by a single bit
- In AVL trees, balancing restricts the difference in heights to at most one
- For red-black trees, we have a different set of rules related to the colours of the nodes

Red-Black Trees

Define a *null path* within a binary tree as any path starting from the root where the last node is not a full node

- Consider the following binary tree:



Red-Black Trees

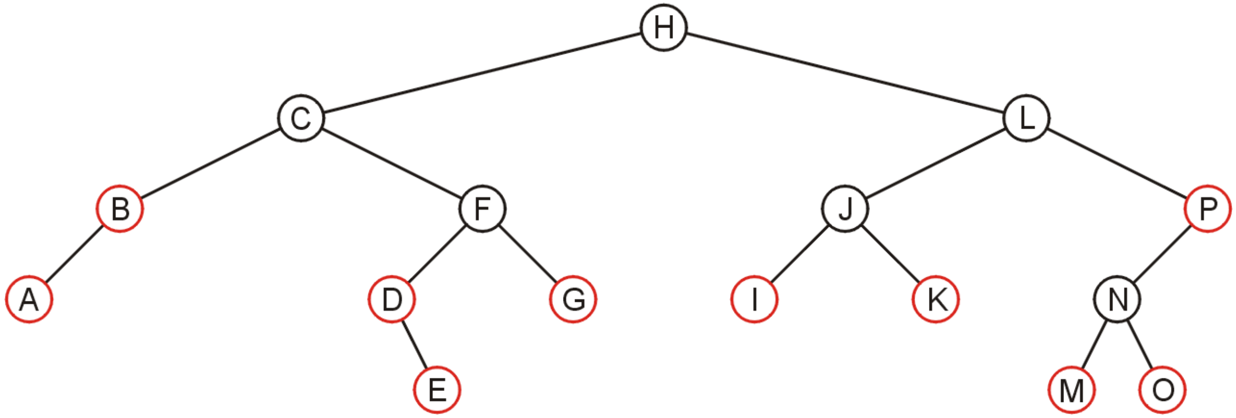
All null paths include:

(H, C, **B**)
(H, C, B, **A**)

(H, C, F, **D**)
(H, C, F, D, **E**)
(H, C, F, **G**)

(H, L, J, **I**)
(H, L, J, **K**)

(H, L, **P**)
(H, L, P, N, **M**)
(H, L, P, N, **O**)



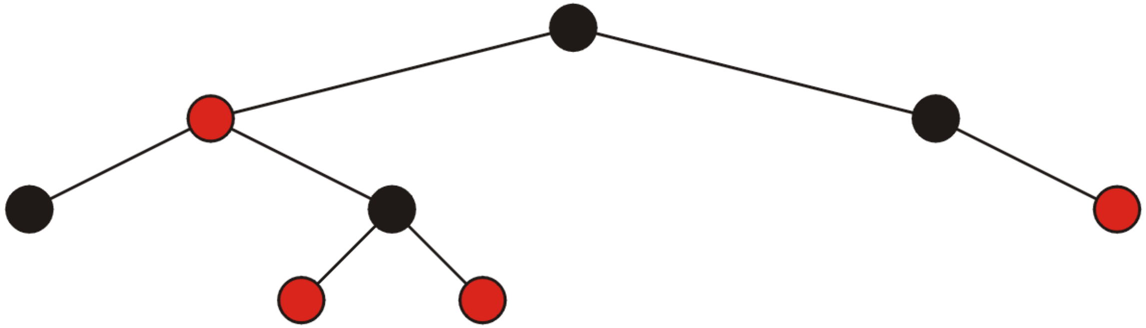
Red-Black Trees

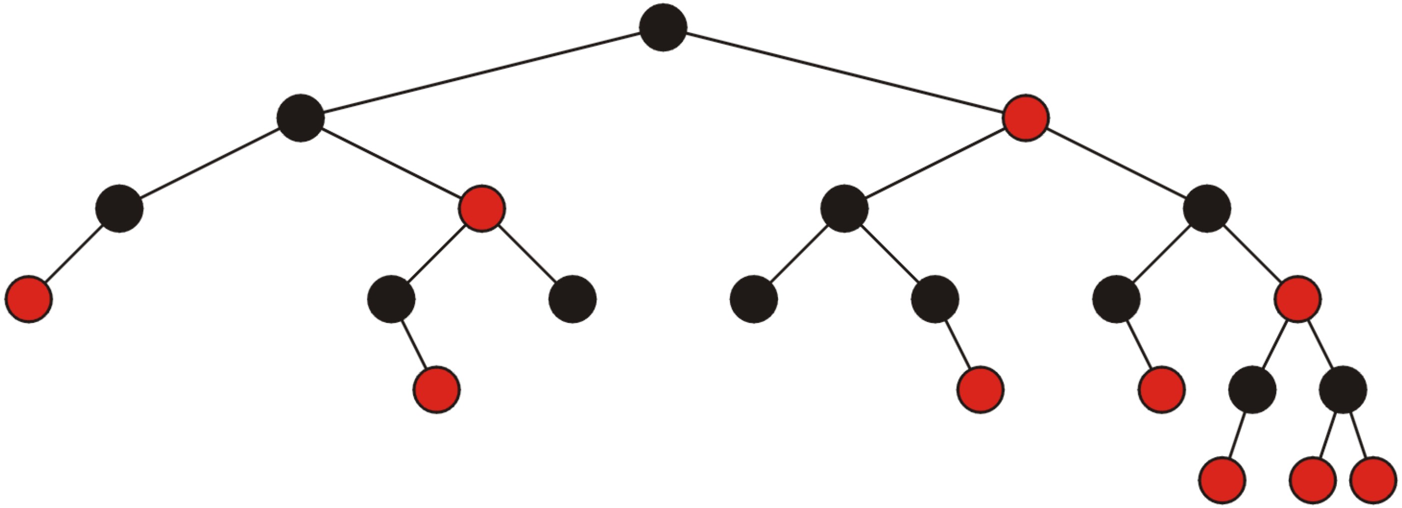
The three rules which define a red-black tree are

1. The root must be black,
2. If a node is red, its children must be black, and
3. Each null path must have the same number of black nodes

Red-Black Trees

These are two examples of red-black trees:





Red-Black Trees

Theorem:

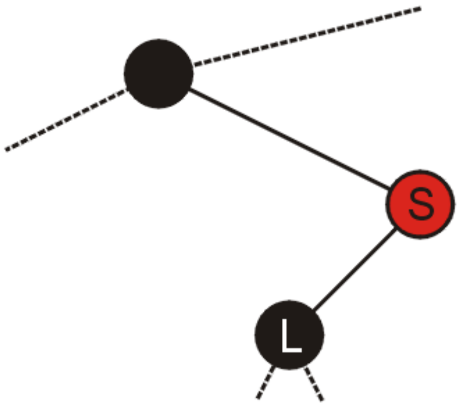
Every red node must be either

- A full node (with two black children), or
- A leaf node

Proof by contradiction:

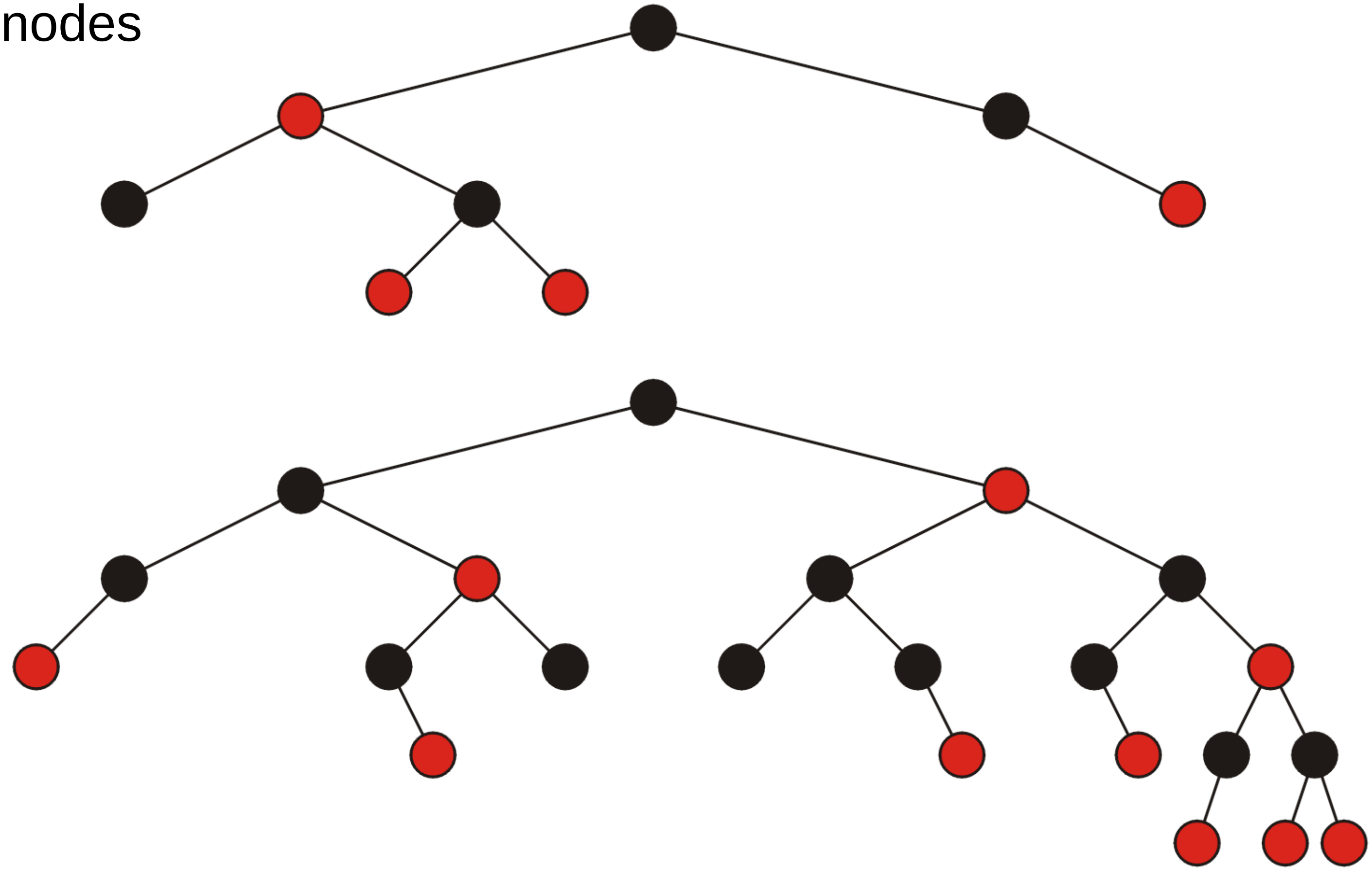
Suppose node S has one child:

- The one child L must be black
- The null path ending at S has k black nodes
- Any null path containing the node L will therefore have at least $k + 1$ black nodes



Red-Black Trees

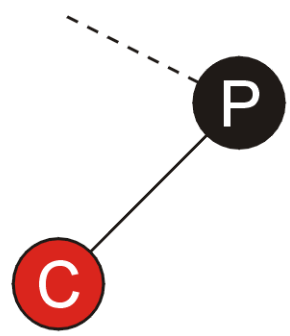
In our two examples, you will note that all red nodes are either full or leaf nodes



Red-Black Trees

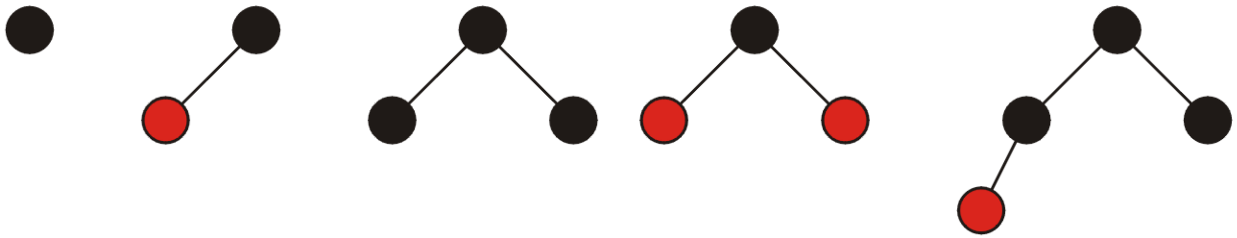
Another consequence is that if a node P has exactly one child:

- The one child must be red,
- The one child must be a leaf node, and
- The node P must be black



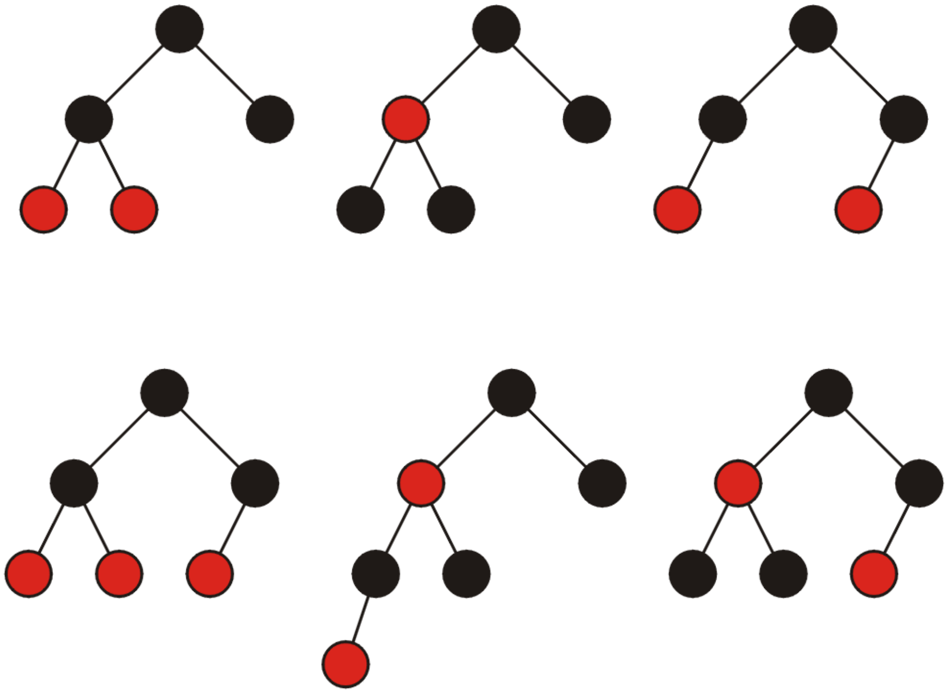
Red-Black Trees

All red-black trees with 1, 2, 3, and 4 nodes:



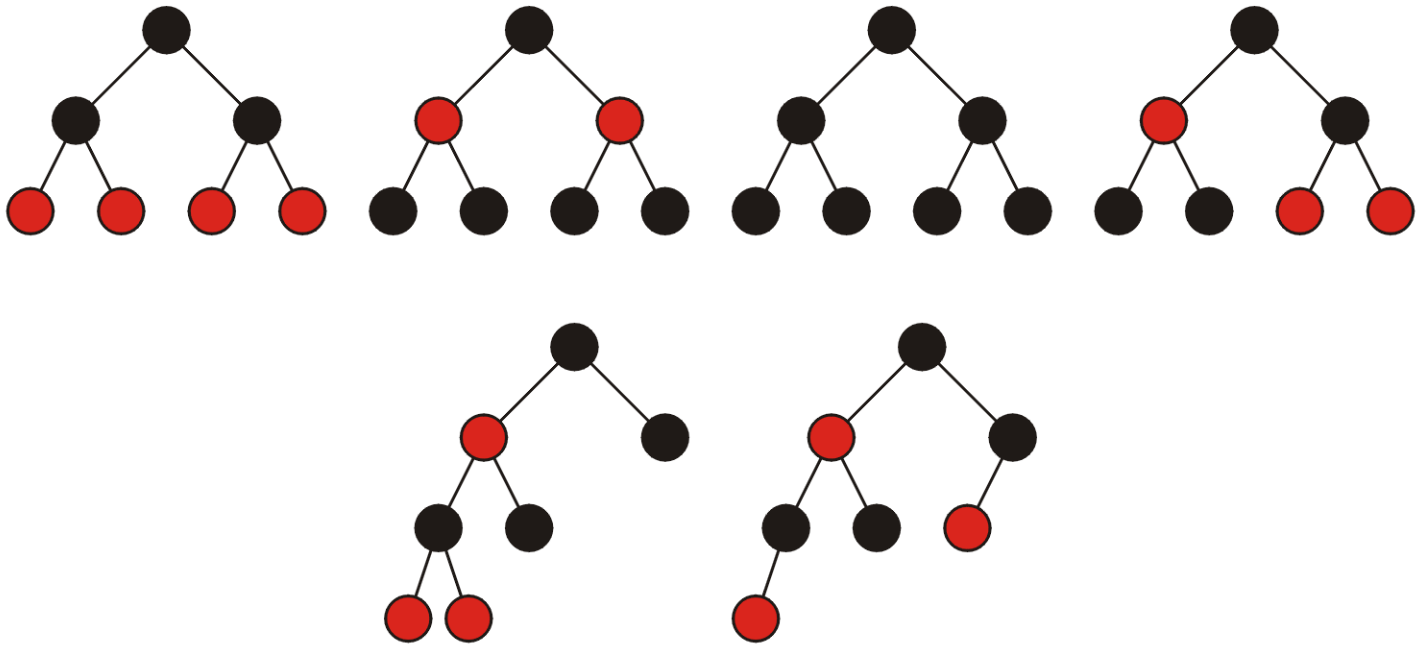
Red-Black Trees

All red-black trees with 5 and 6 nodes:



Red-Black Trees

All red-black trees with seven nodes—most are shallow:



Red-Black Trees

Every perfect tree is a red-black tree if each node is coloured black

A complete tree is a red-black tree if:

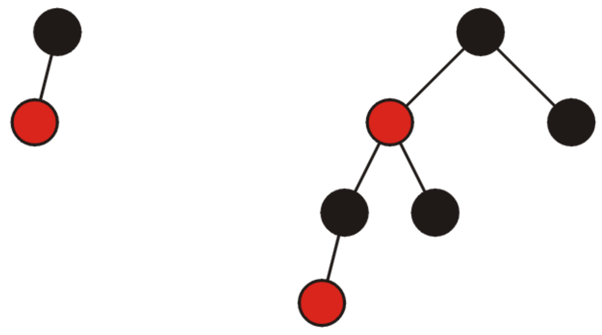
- each node at the lowest depth is coloured red, and
- all other nodes are coloured black

What is the worst case?

Red-Black Trees: Worst Case

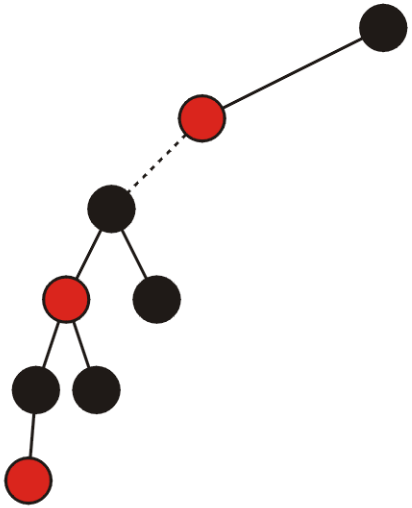
Any worst-case red-black tree must have an alternating red-black pattern down one side

The following are the worst-case red-black trees with 1 and 2 black nodes per null path (*i*.*e*., heights 1 and 3)



Red-Black Trees: Worst Case

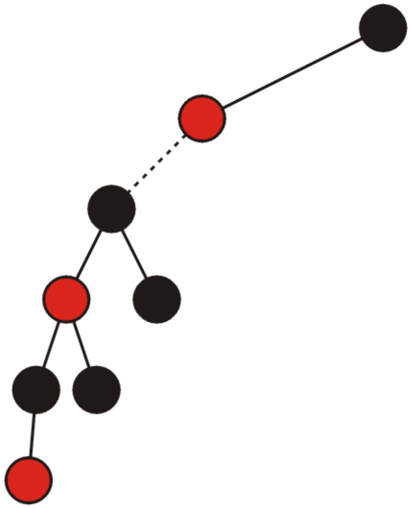
To create the worst-case for paths with 3 black nodes per path, start with a black and red node and add the previous worst-case for paths with 2 nodes



Red-Black Trees: Worst Case

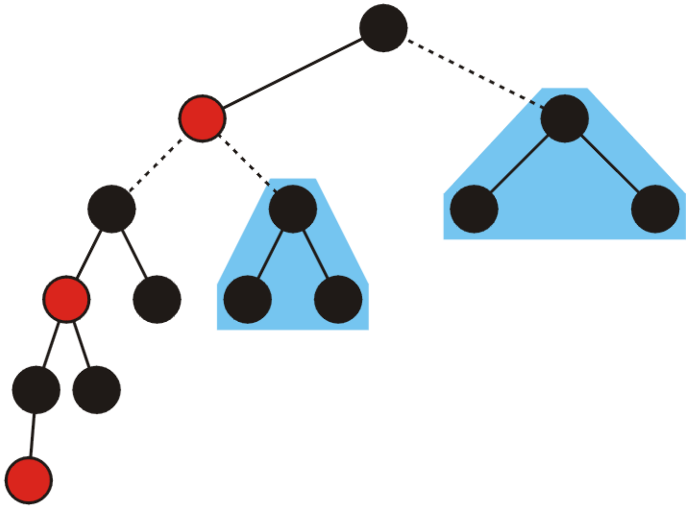
This, however, is not a red-black tree because the two top nodes do not have paths with three black nodes

- To solve this, add the optimal red-black trees with two black nodes per path



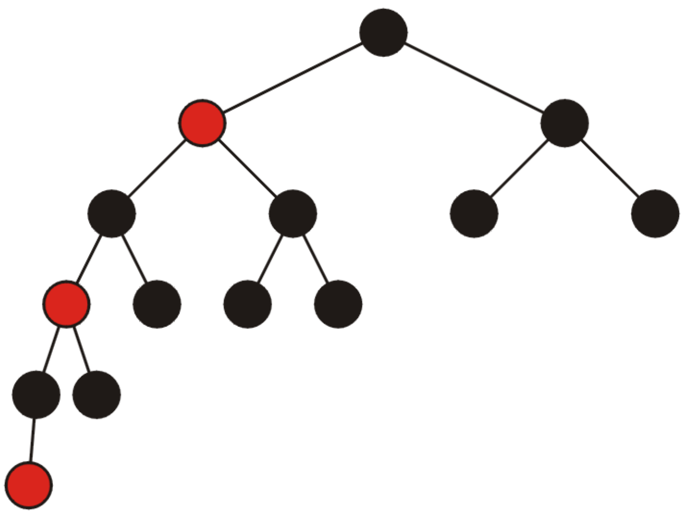
Red-Black Trees: Worst Case

That is, add two perfect trees with height 1 to each of the missing sub-trees



Red-Black Trees: Worst Case

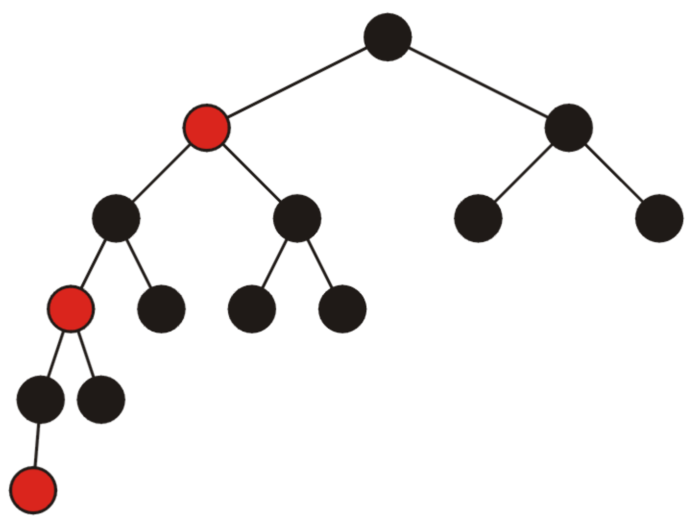
Thus, we have the worst-case for a red-black tree with three black nodes per path (or a red-black tree of height 5)



Red-Black Trees: Worst Case

Note that the left sub-tree of the root has height 4 while the right has height 1

– Thus suggests that AVL trees may be better in maintaining "balance"



Insertions

We will consider two types of insertions:

- bottom-up (insertion at the leaves), and
- top-down (insertion at the root)

The first will be instructional and we will use it to derive the second case

Bottom-Up Insertions

After an insertion is performed, we must satisfy all the rules of a red-black tree:

#1. The root must be black,

#2. If a node is red, its children must be black, and

#3. Each path from a node to any of its descendants which are is not a full node (*i.e.*, two children) must have the same number of black nodes

#1 and #2 are local: they affect a node and its neighbours

#3 is global: adding a new black node anywhere will cause all of its ancestors to become unbalanced

Bottom-Up Insertions

Thus, when we add a new node, we will add **a red node**

- Which breaks the local rule
- But not breaking the global rule

We will then travel up the tree to the root, while fixing the requirement #1 and #2

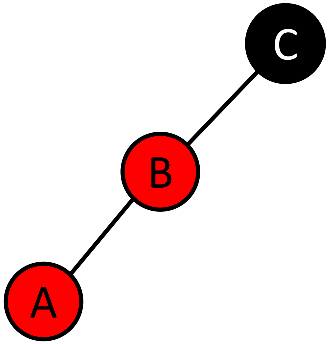
Bottom-Up Insertions

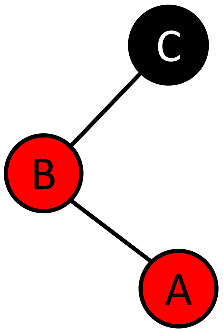
Case A. If the parent of the inserted “red” node is already black,

- nothing to be done.

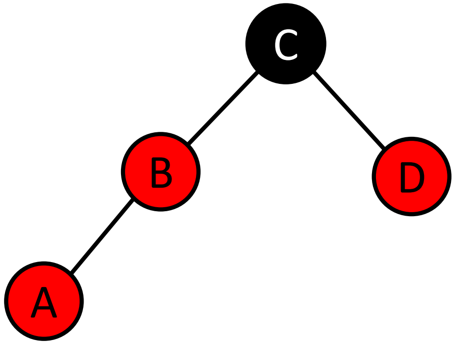
Case B. If the parent is red, then we need to fix:

– Case #B1: the grandparent has one red child





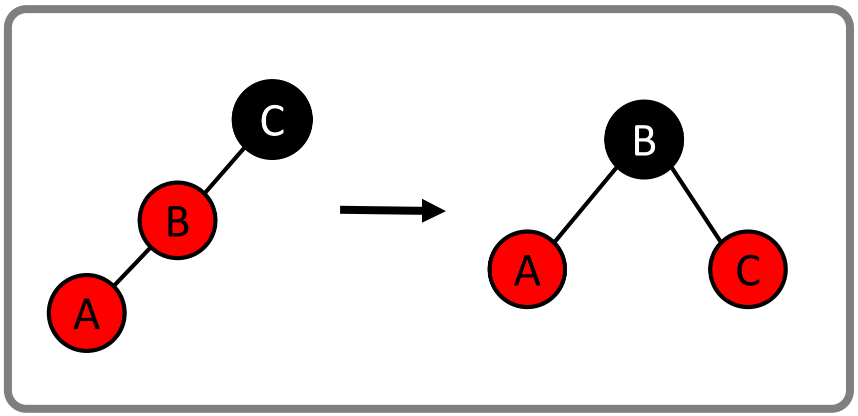
– Case #B2: the grandparent has two red children

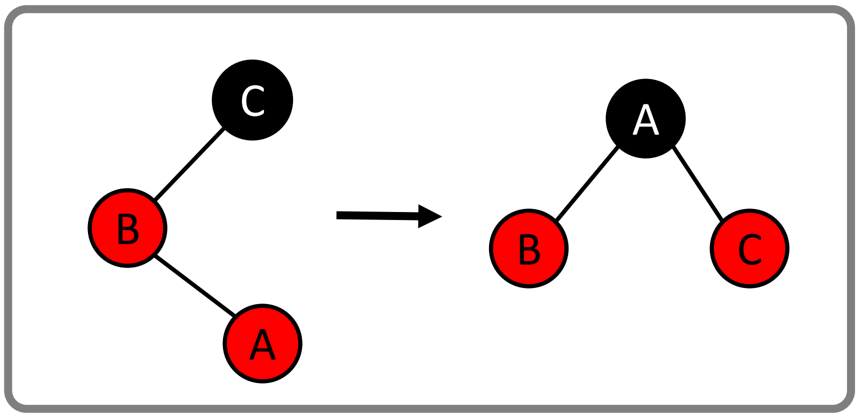


Bottom-Up Insertions: Case #B1

Case #B1 can be fixed with **rotation.**

Example: Inserting A

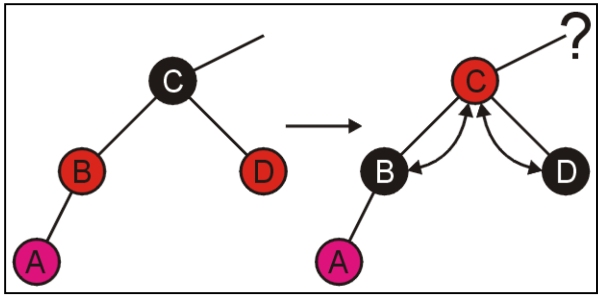




Consequently, we are finished...

Bottom-Up Insertions: Case #B2

Case #B2 seems to be fixed by just **swapping the colours**:



However, this may have a problem:

- **Red-Red pair**: C is red and C's parent may be red as well.

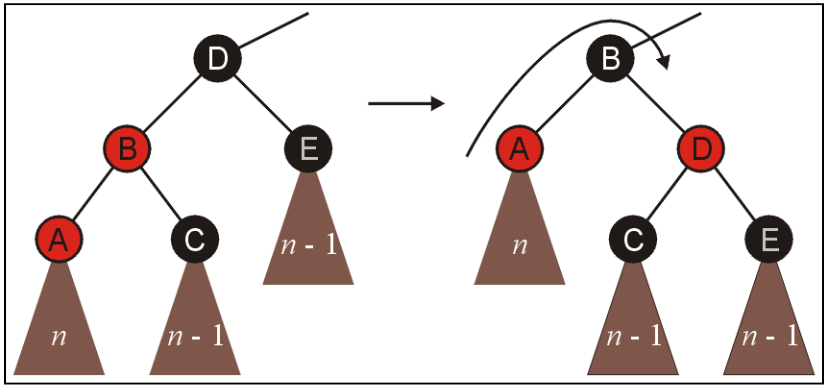
After swapping the colours, there can be yet another two cases:

- **Case #B2a**. C's grand parent has one red child
- **Case #B2b**. C's grand parent has two red children

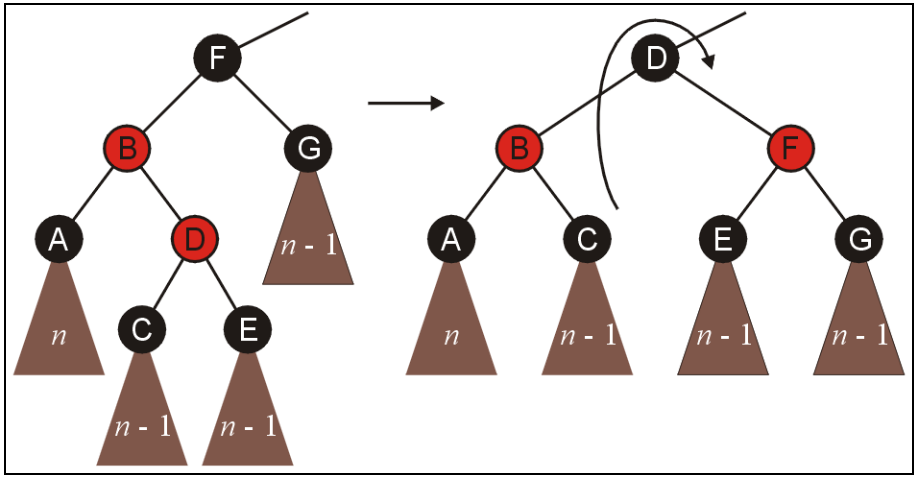
Bottom-Up Insertions: Case #B2a

Case #B2a. If the grand parent had one red child

➔ Perform similar **rotations** as we have done before.



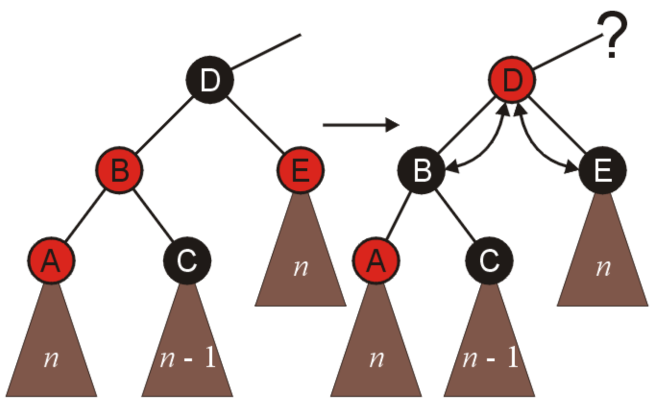
The grand parent (D) has only one red child (B)



The grand parent (F) has only one red child (B)

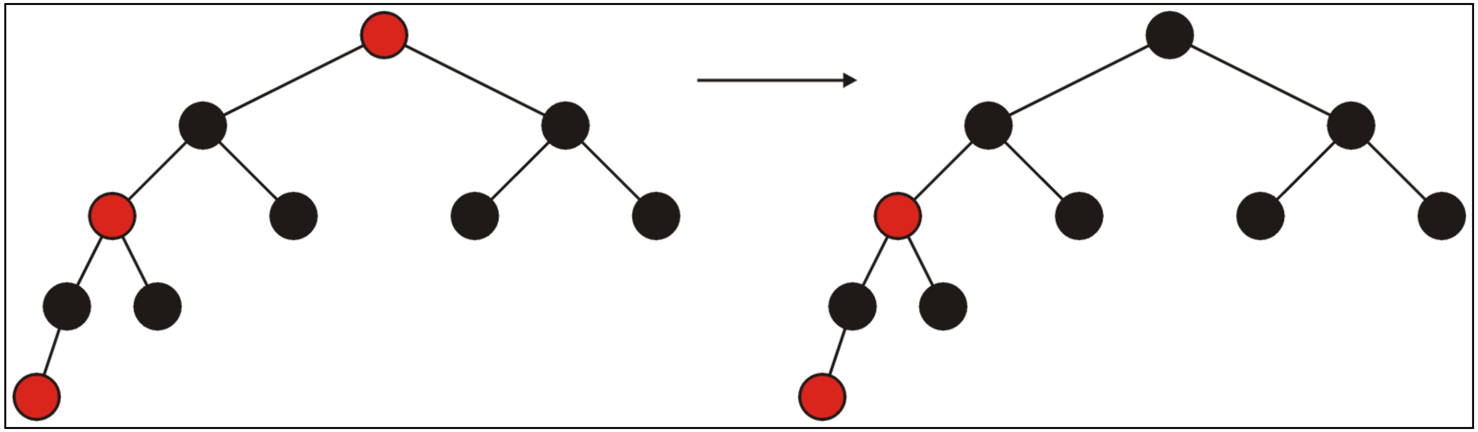
Bottom-Up Insertions: Case #B2b

Case #B2b. If both children of the grandparent are red
➔ we **swap colours**, and then recurs back to the root



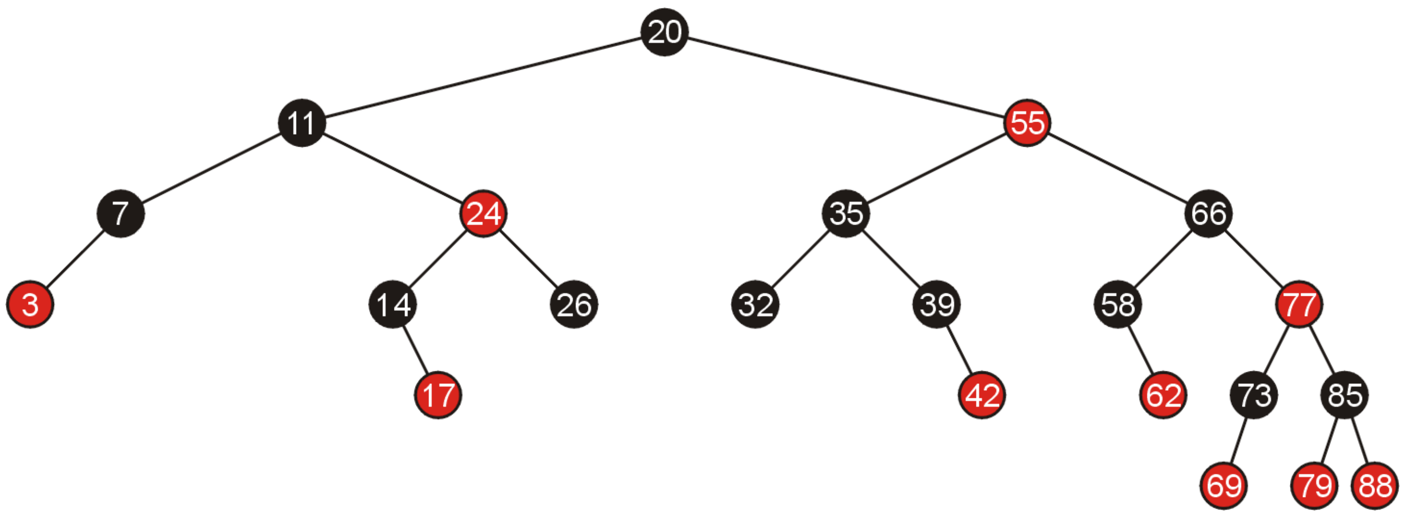
Bottom-Up Insertions: Case #B2b

If, at the end, the root is red, it can be coloured black



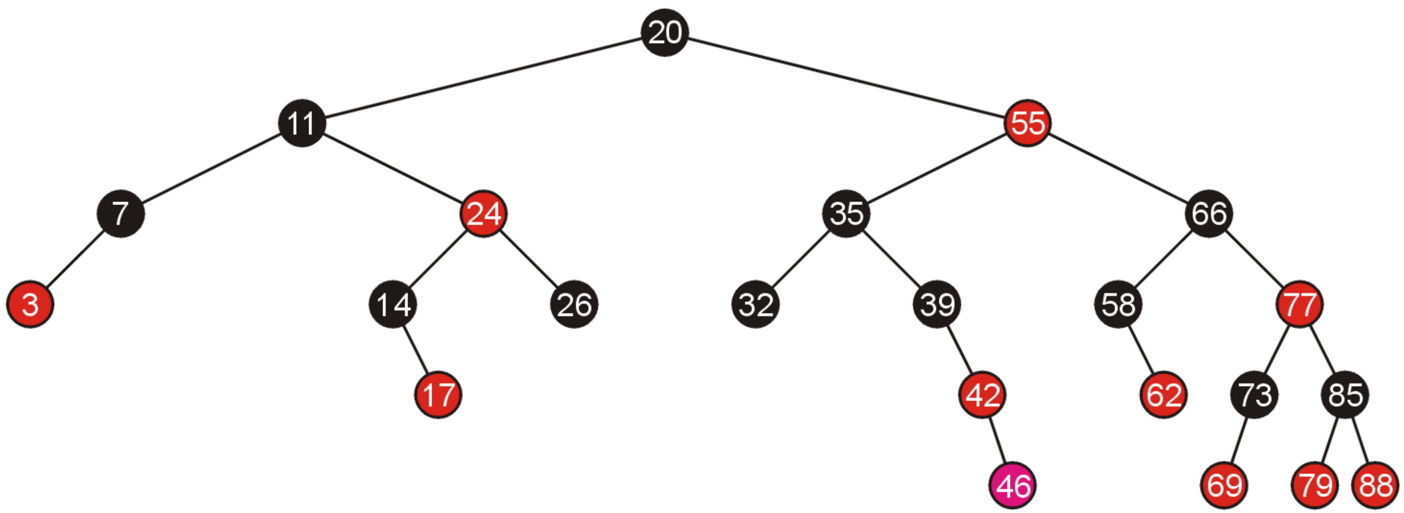
Examples of Insertions

Given the following red-black tree, we will make a number of insertions



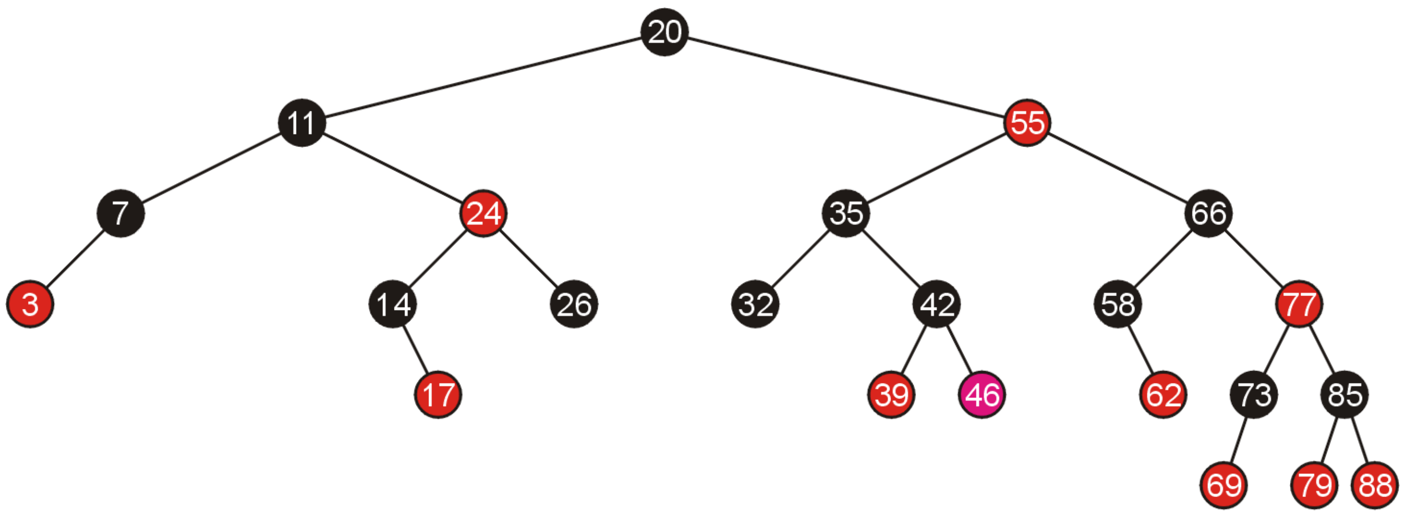
Examples of Insertions: Insert 46

Adding 46 creates a red-red pair which can be corrected with a single **rotation**



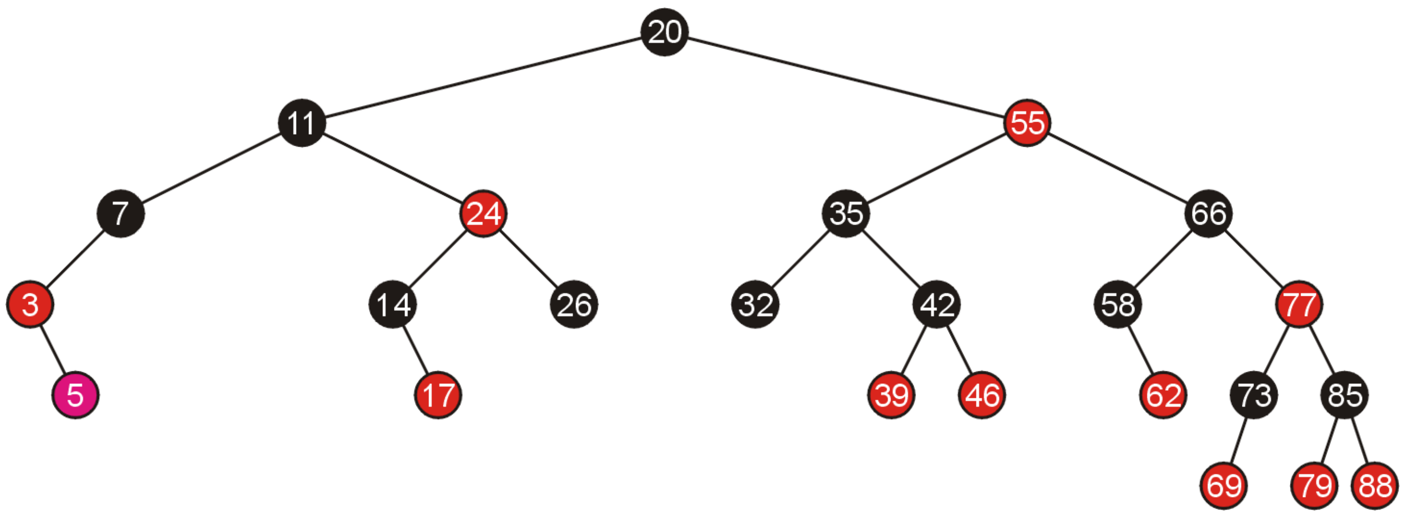
Examples of Insertions: Insert 46

Because the pivot is still black, we are finished



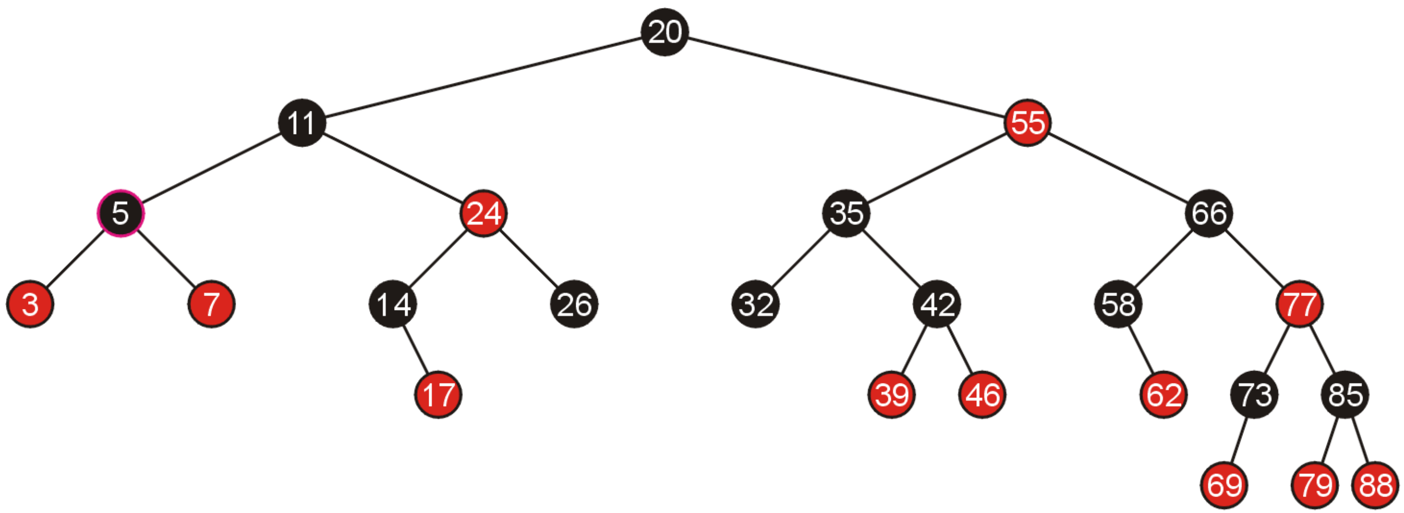
Examples of Insertions: Insert 5

Similarly, adding 5 requires a single **rotation**



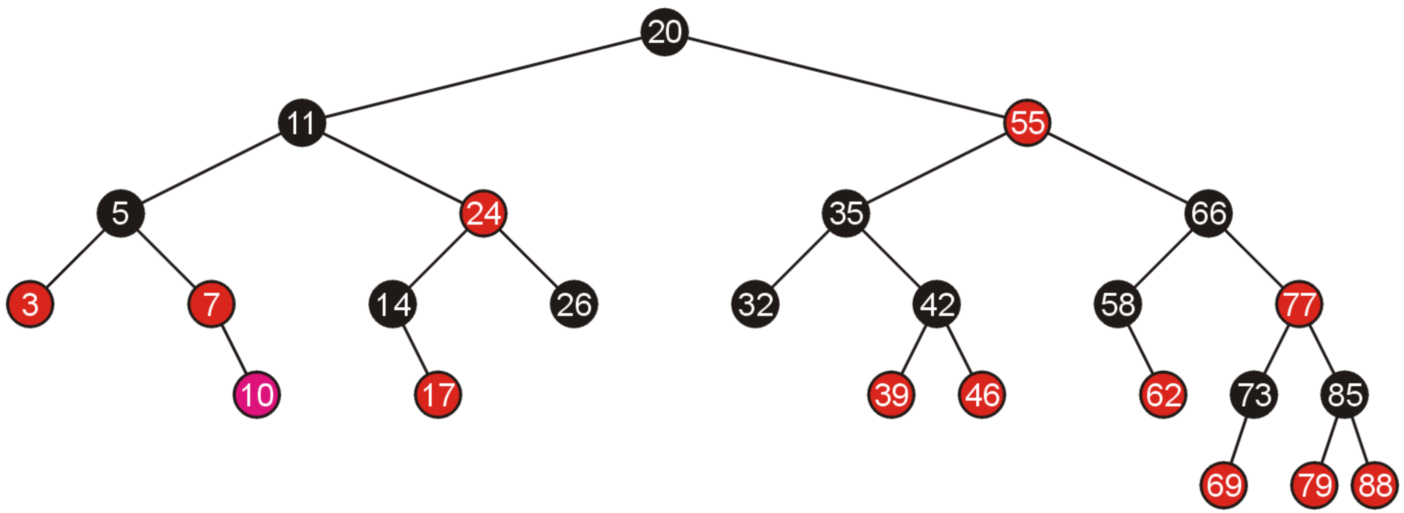
Examples of Insertions: Insert 5

Which again, does not require any additional work



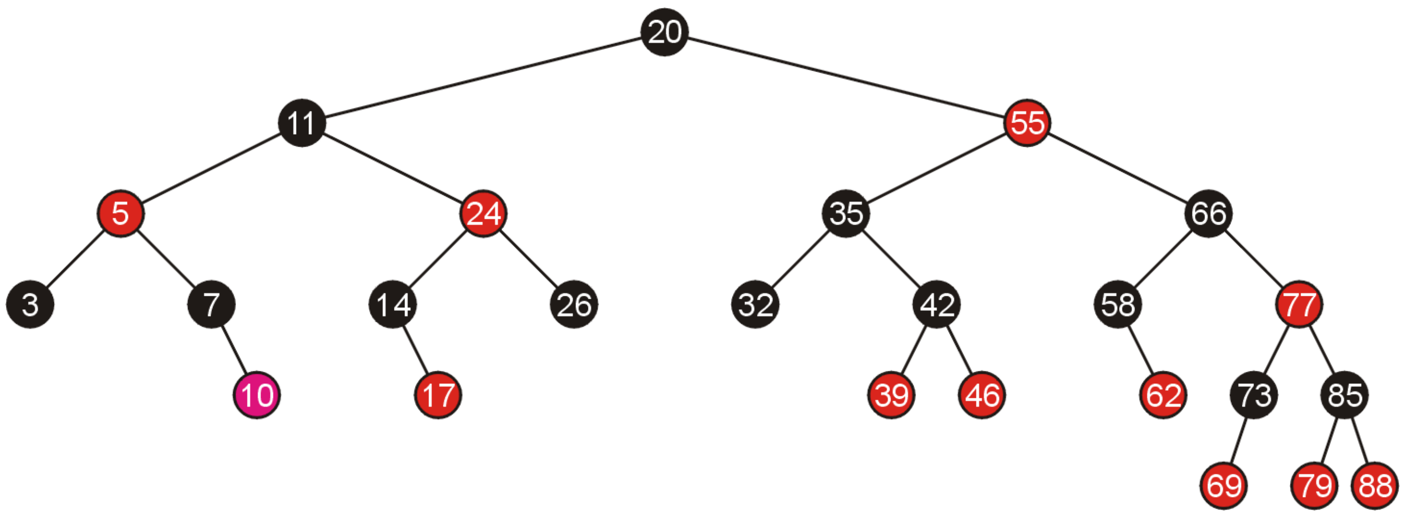
Examples of Insertions: Insert 10

Adding 10 allows us to simply **swap the colour** of the grand parent and the parent and the parent's sibling



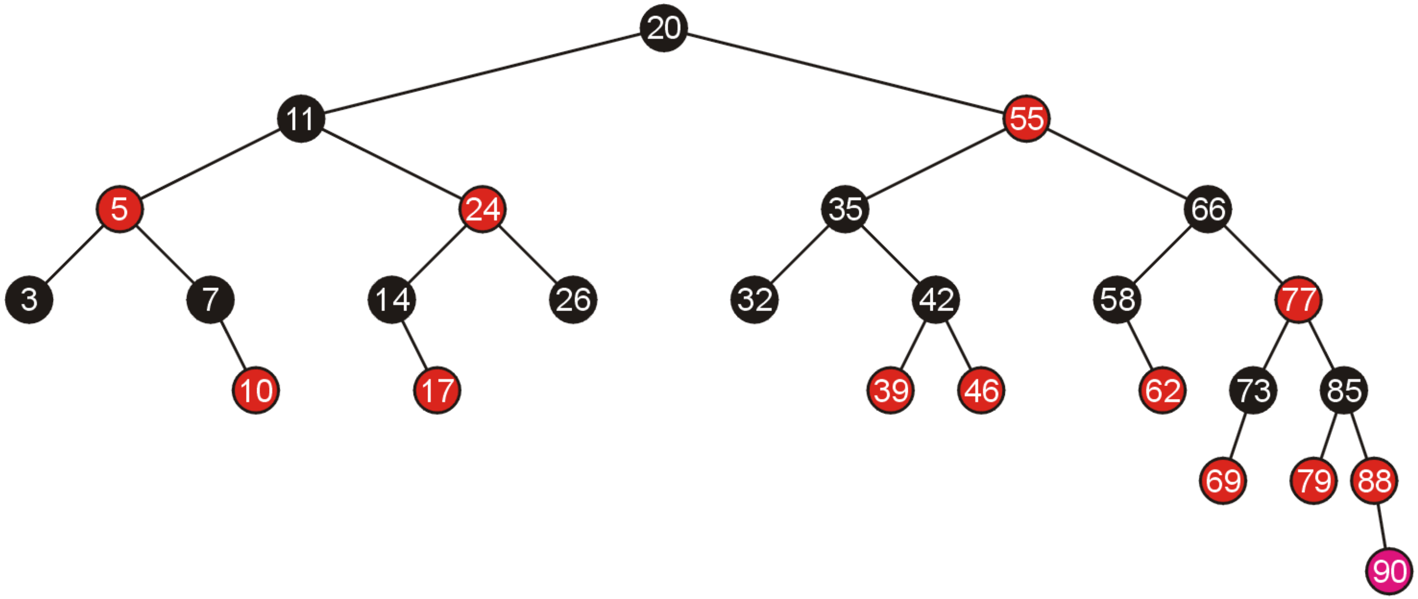
Examples of Insertions: Insert 10

Because the parent of 5 is black, we are finished



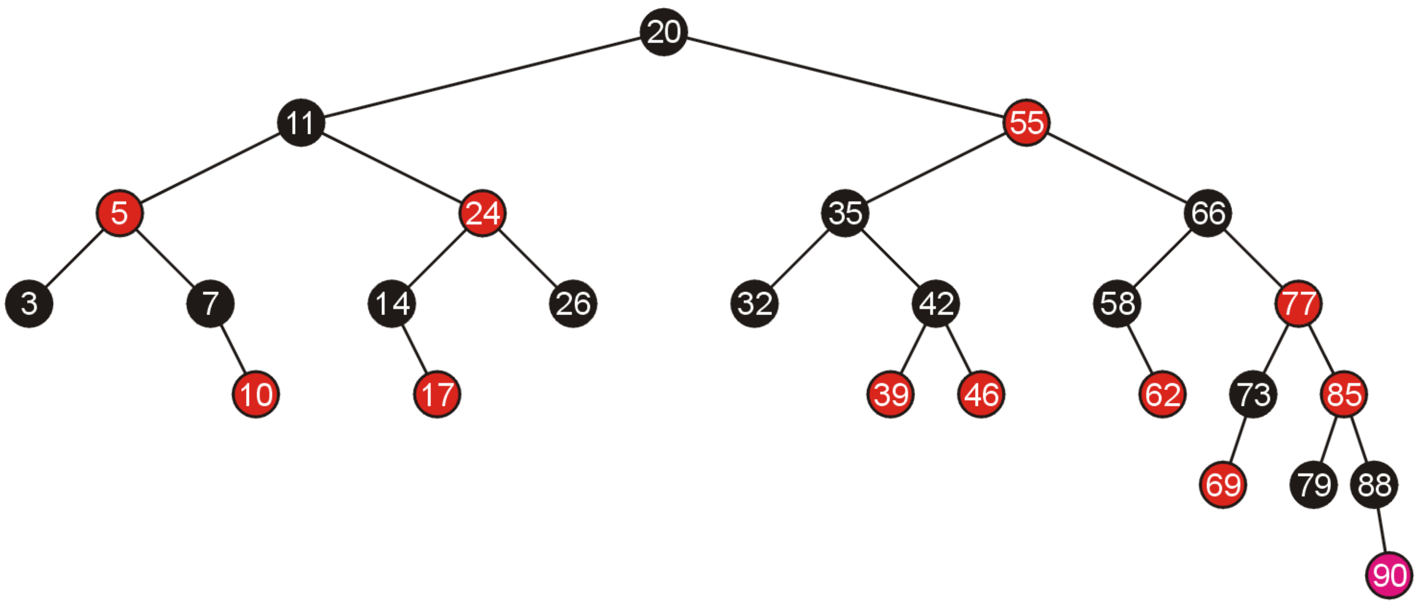
Examples of Insertions: Insert 90

Adding 90 again requires us to **swap the colours** of the grandparent and its two children



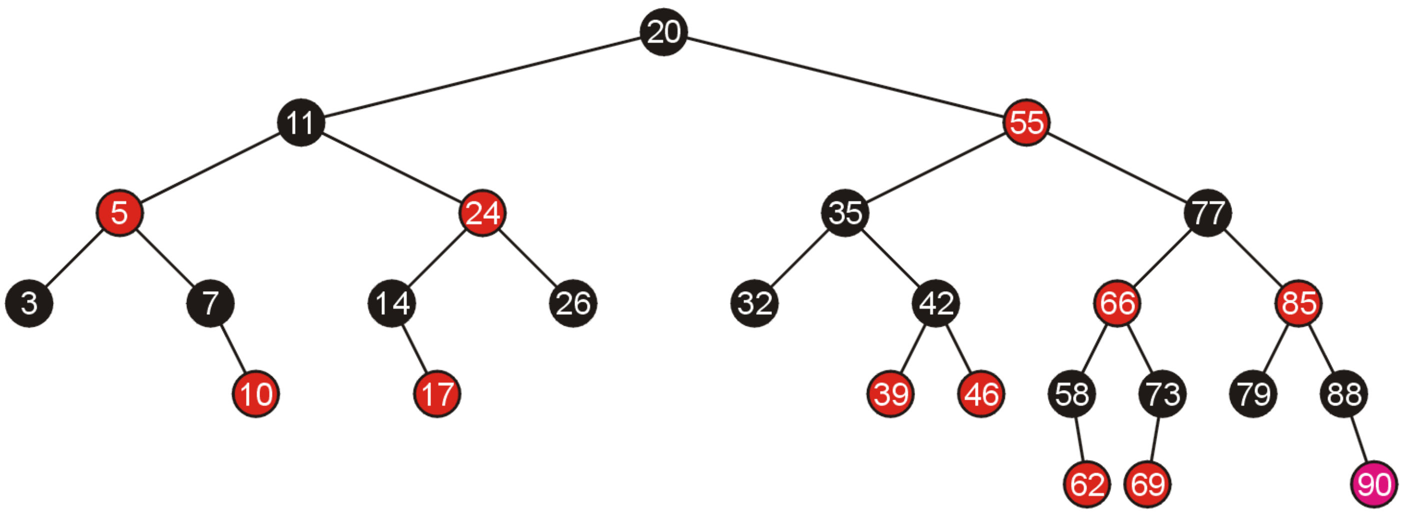
Examples of Insertions: Insert 90

This causes a red-red parent-child pair, which **now requires a rotation**



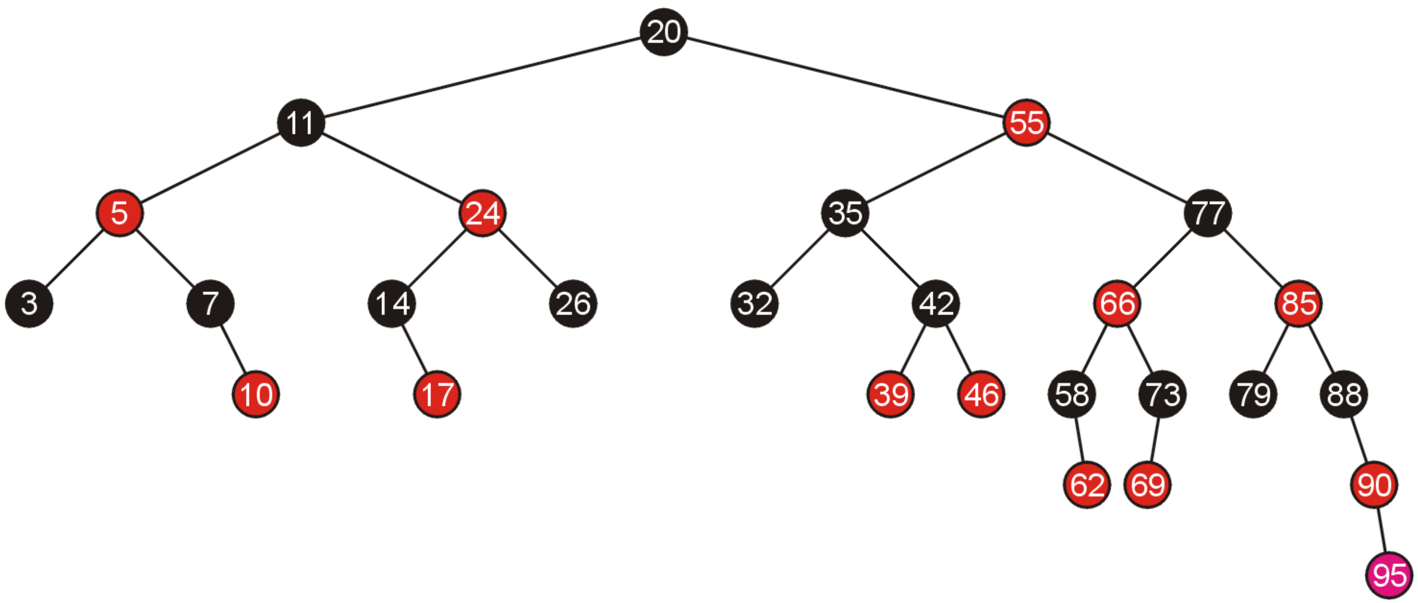
Examples of Insertions: Insert 90

A rotation does not require any subsequent modifications, so we are finished



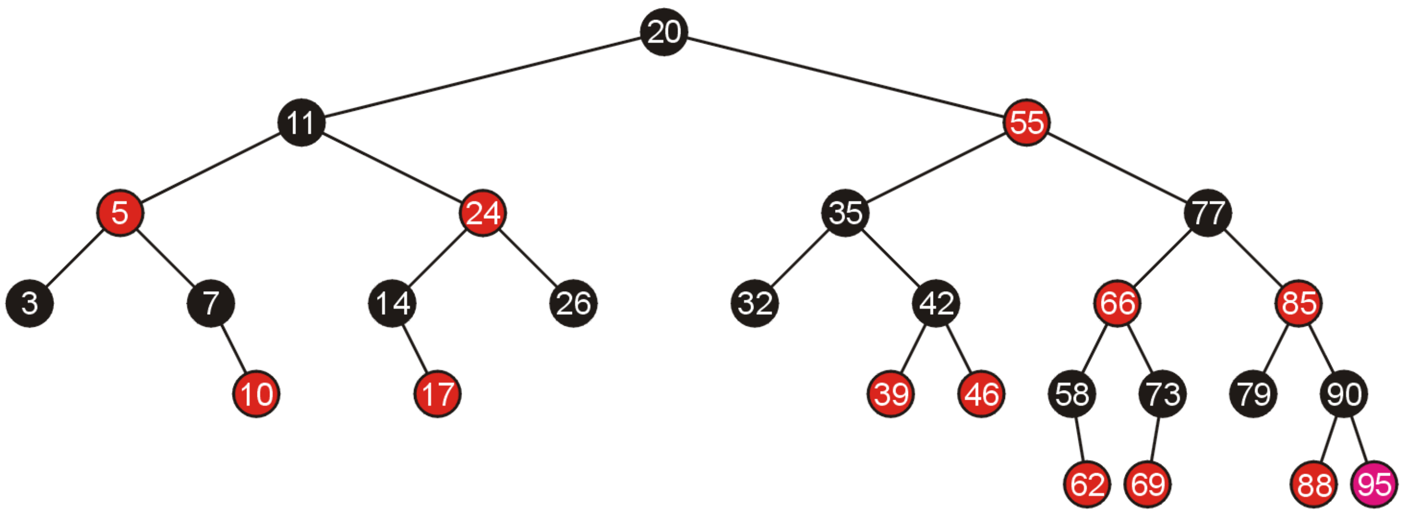
Examples of Insertions: Insert 95

Inserting 95 requires a single **rotation**



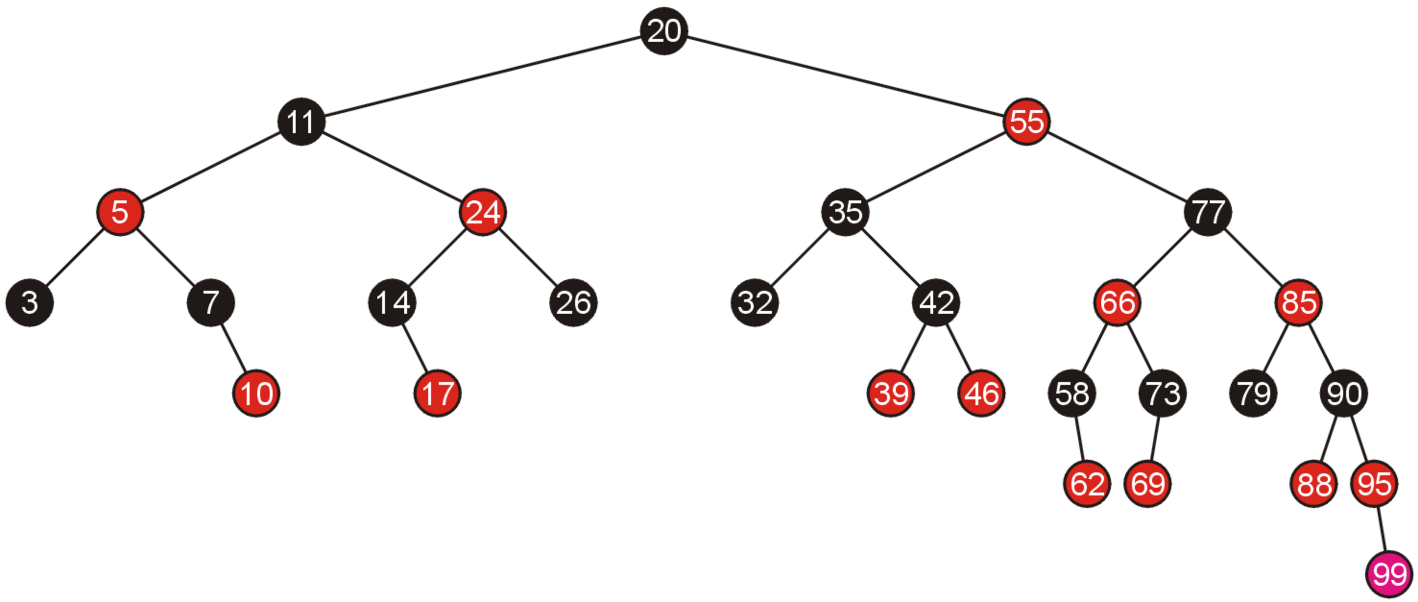
Examples of Insertions: Insert 95

And consequently, we are finished



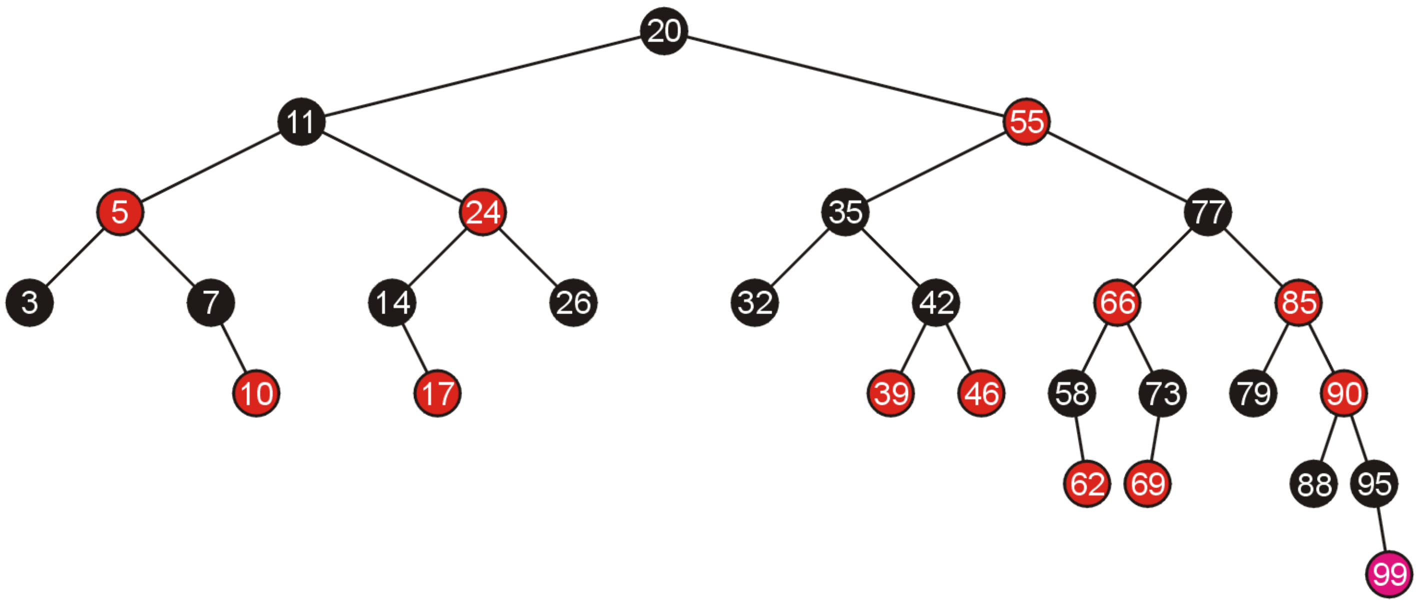
Examples of Insertions: Insert 99

Adding 99 requires us to **swap the colours** of its grandparent and the grandparent's children



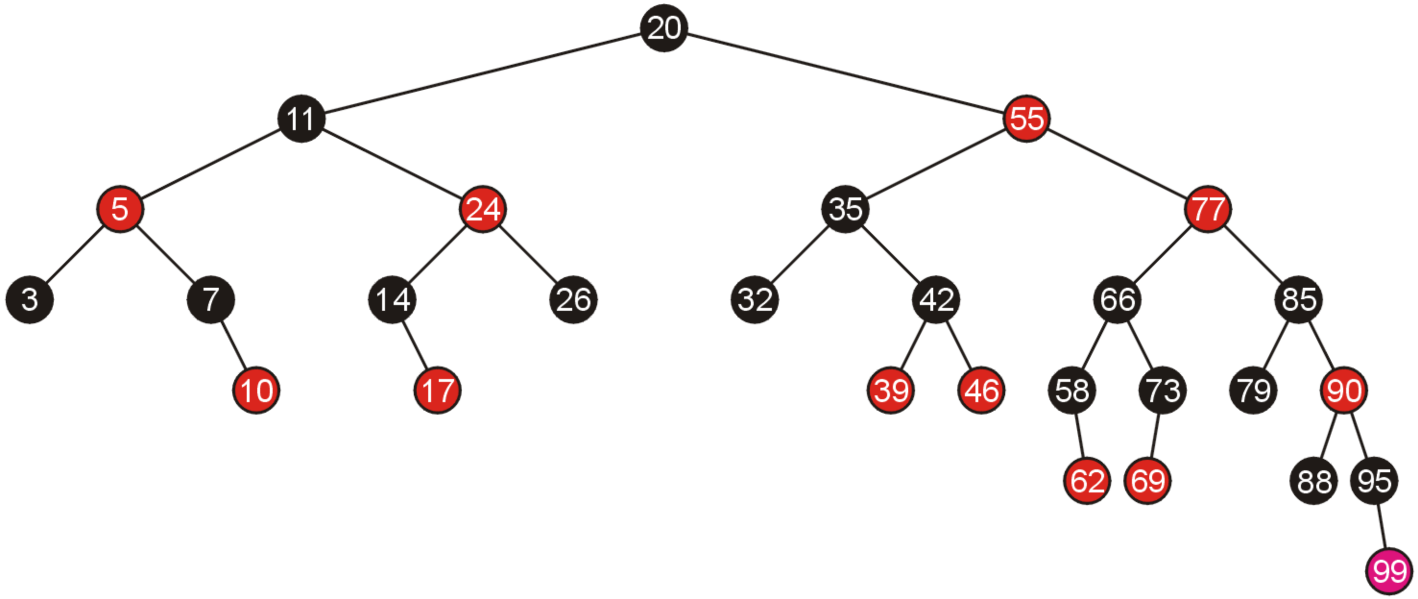
Examples of Insertions: Insert 99

This causes another red-red child-parent conflict between 85 and 90 which must be fixed, again by **swapping colours**



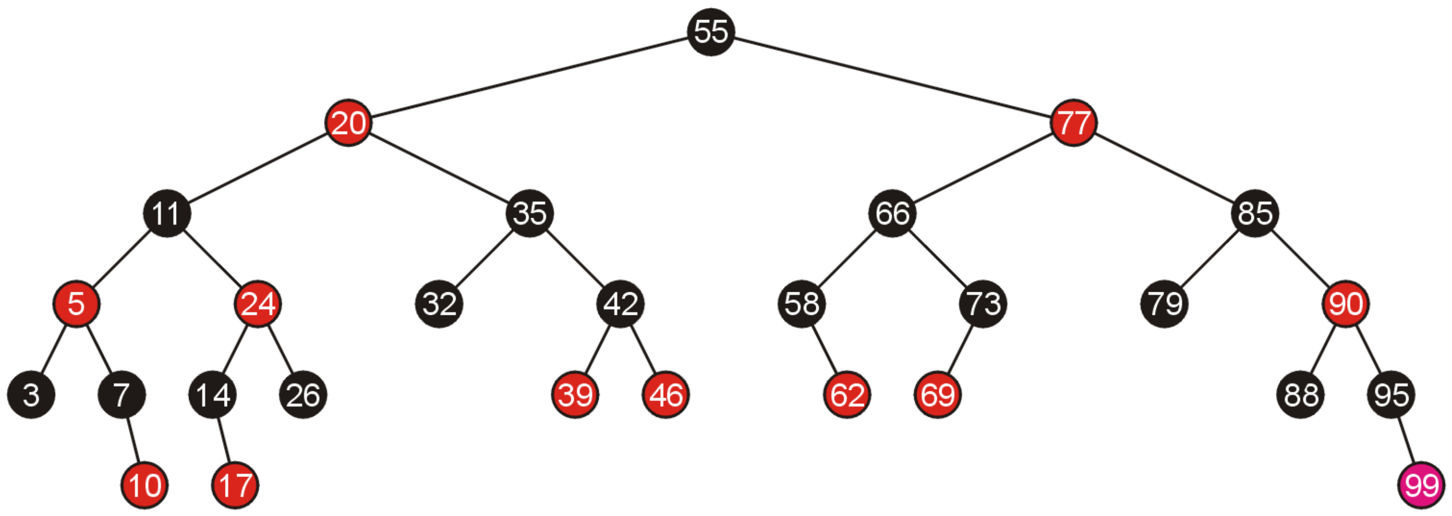
Examples of Insertions: Insert 99

This results in another red-red parent-child conflict, this time, requiring a **rotation**



Examples of Insertions: Insert 99

Thus, the rotation solves the problem



Top-Down Insertions and Deletions

Bottom-up insertion:

1) search the tree for the appropriate location
2) insert a red node
3) recurs back to the root correcting any problems

– This is similar to AVL trees

Top-down insertion/deletion: With red-black trees, it is possible to perform both insertions and deletions strictly by starting at the root, but not requiring the recurs back to the root

Top-Down Insertions

Observation from bottom-up insertions:

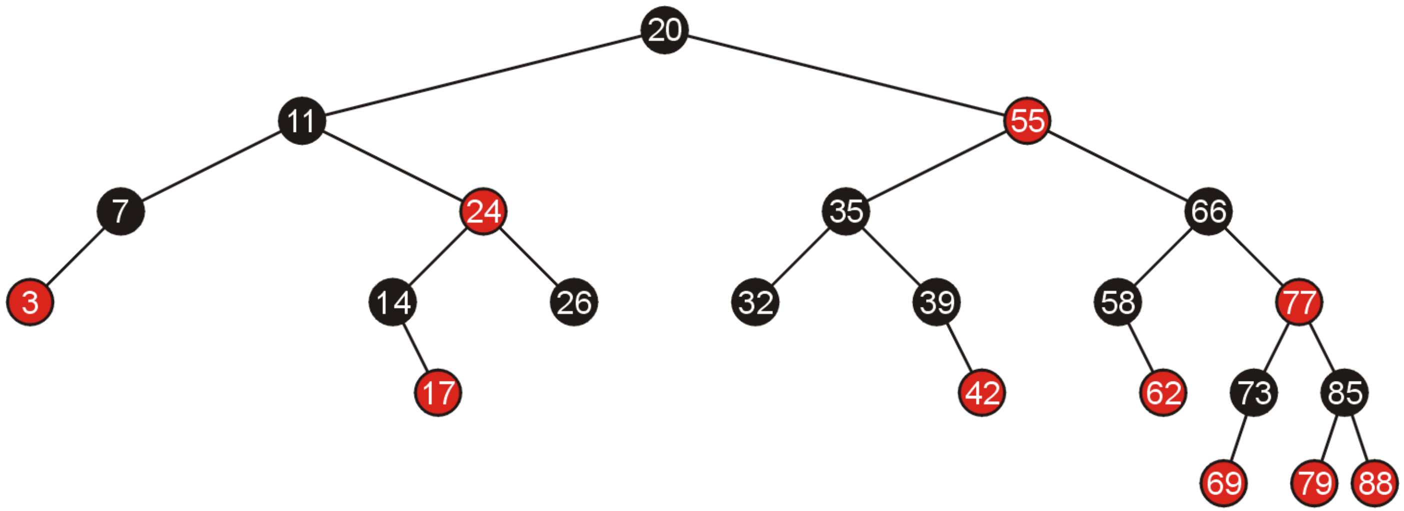
- **Rotations** (Case #B1/B2a) **do not require recursive steps**. These cases had a one red child node
- **Swapping** (Case #B2/B2b) may **require recursive corrections**. These cases had two red children node

Ideas for top-down insertions:

- Automatically swap the colours of any black node with two red children while moving down from the root.
- This may require at most one rotation at the parent of the now-red node

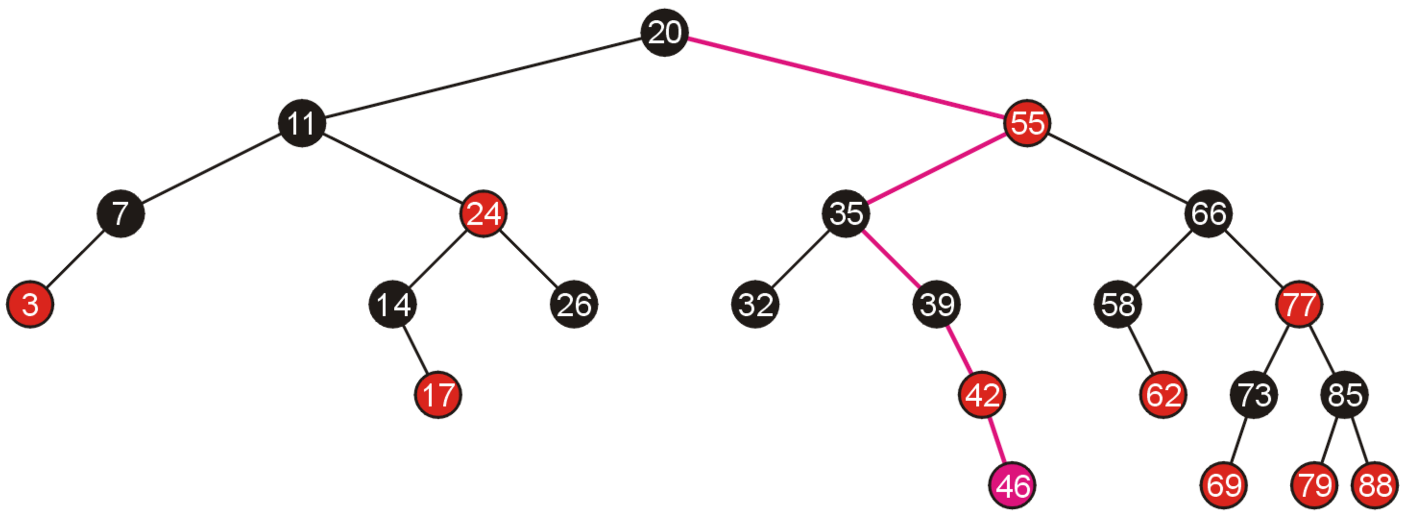
Examples of Top-Down Insertions

Now we will perform top-down insertions.



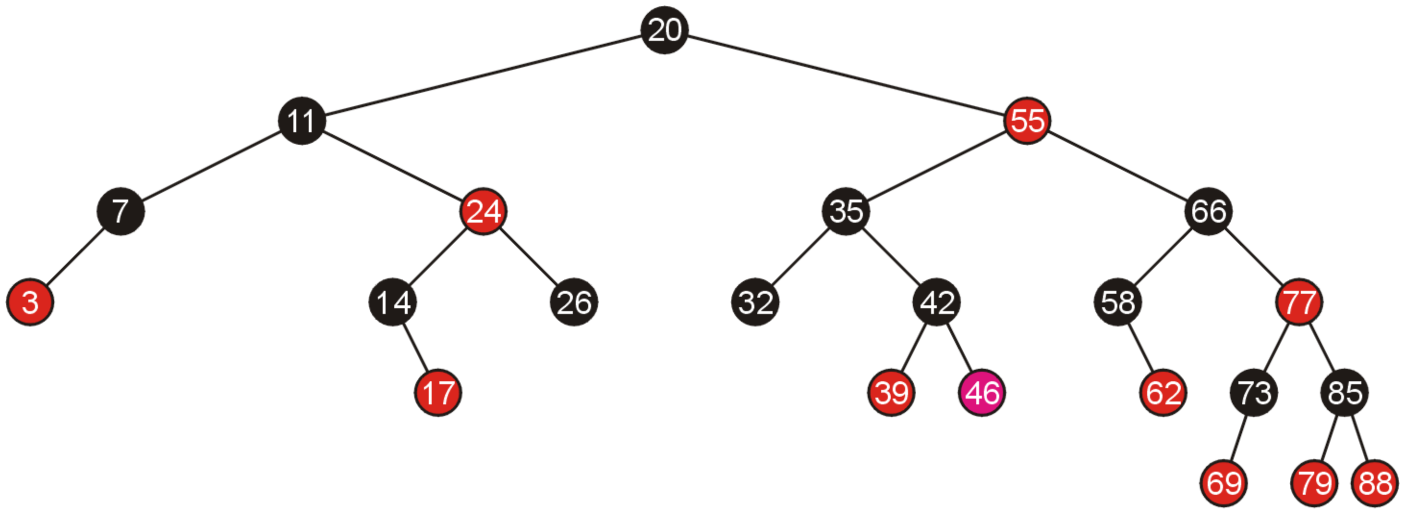
Examples of Top-Down Insertions: Insert 46

Adding 46 does not find any (necessarily black) parent with two red children



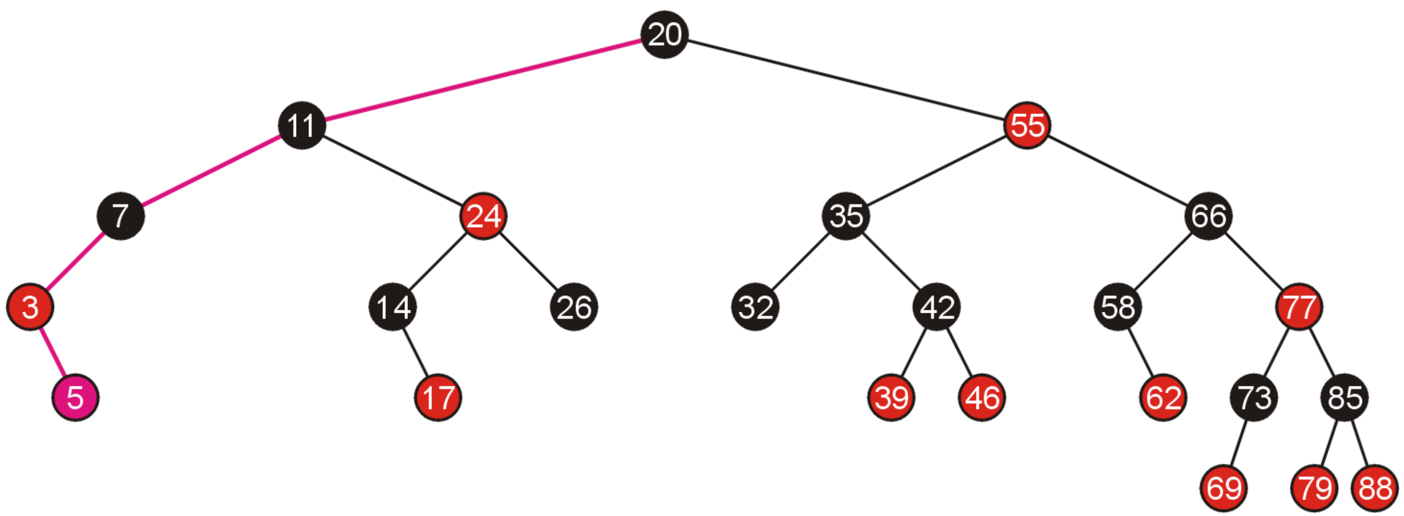
Examples of Top-Down Insertions: Insert 46

However, it does require one **rotation** at the end



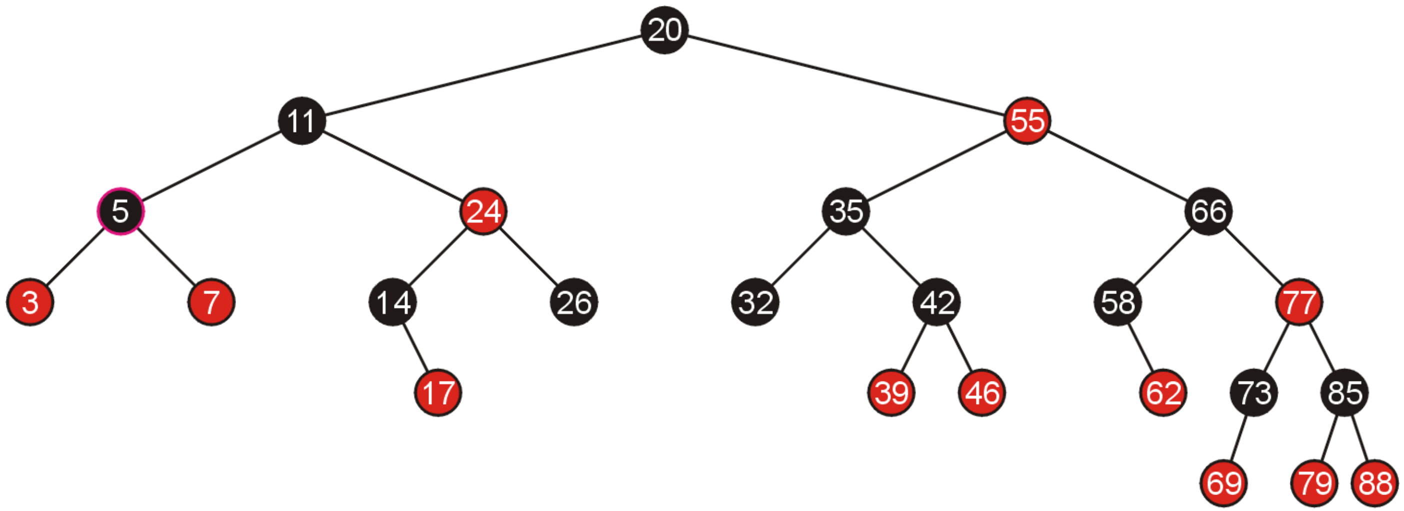
Examples of Top-Down Insertions: Insert 5

Similarly, adding 5 does not meet any parent with two red children:



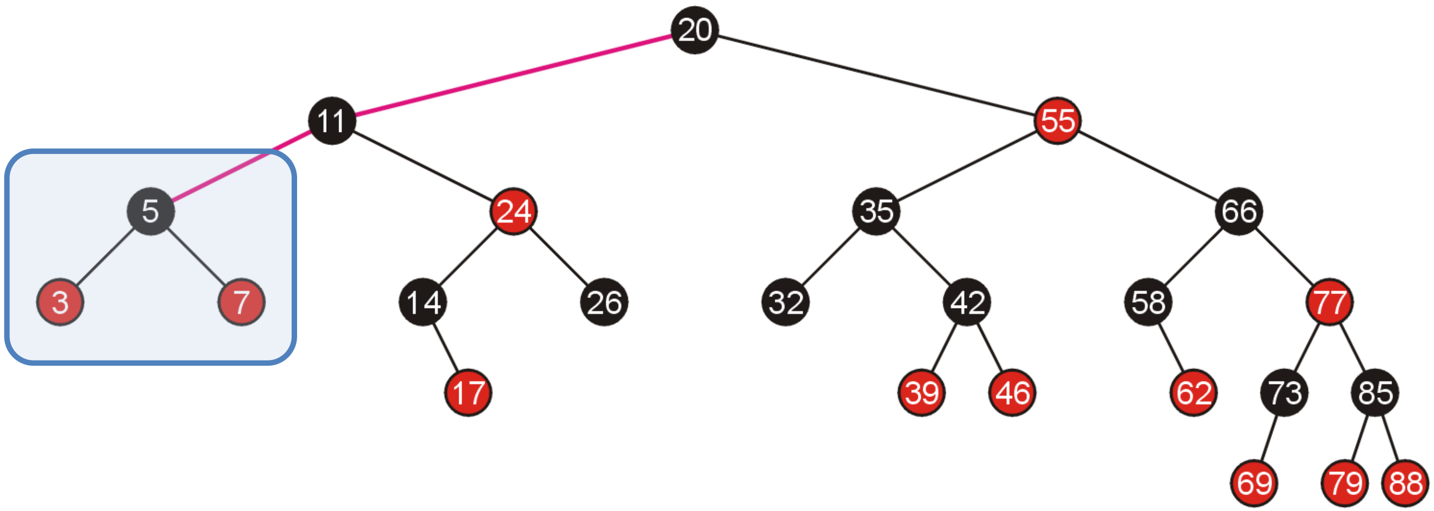
Examples of Top-Down Insertions: Insert 5

A **rotation** solves the last problem



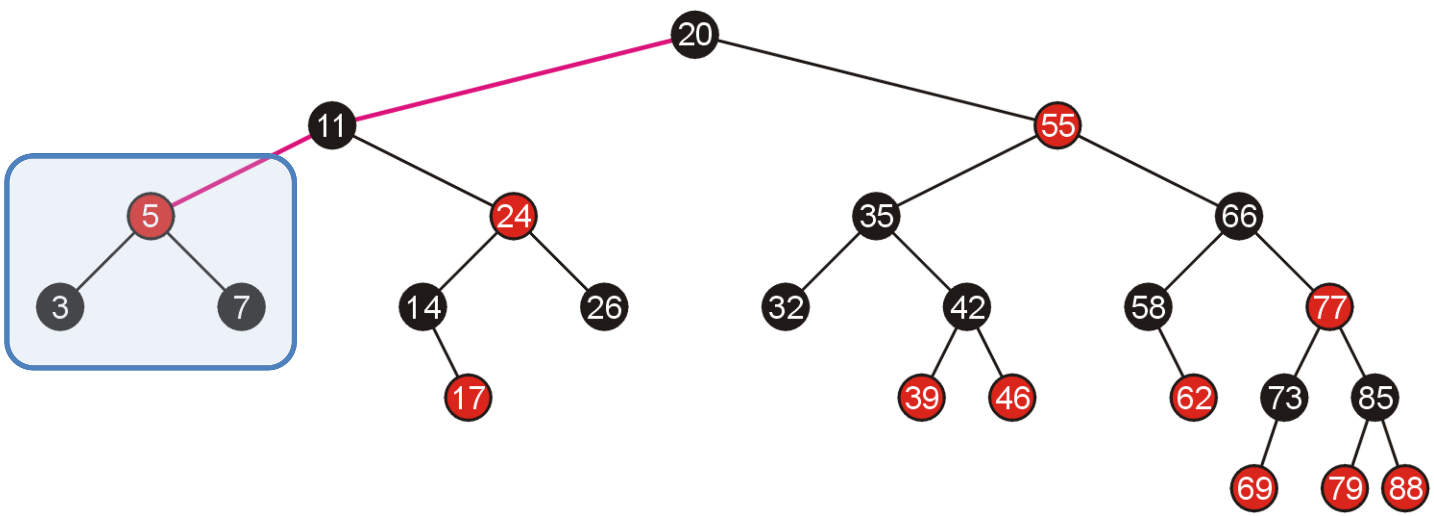
Examples of Top-Down Insertions: Insert 10

To insert 10, we can spot that node 5 has two red children



Examples of Top-Down Insertions: Insert 10

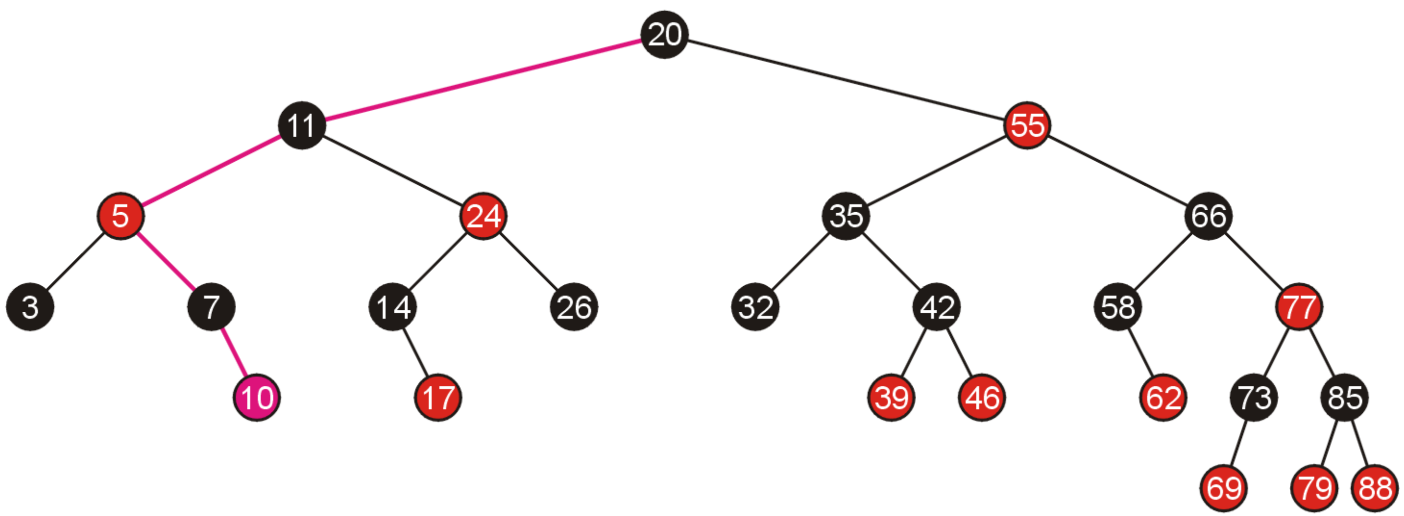
We **swap the colours**, and this does not cause a problem between 5 and 11



Examples of Top-Down Insertions: Insert 10

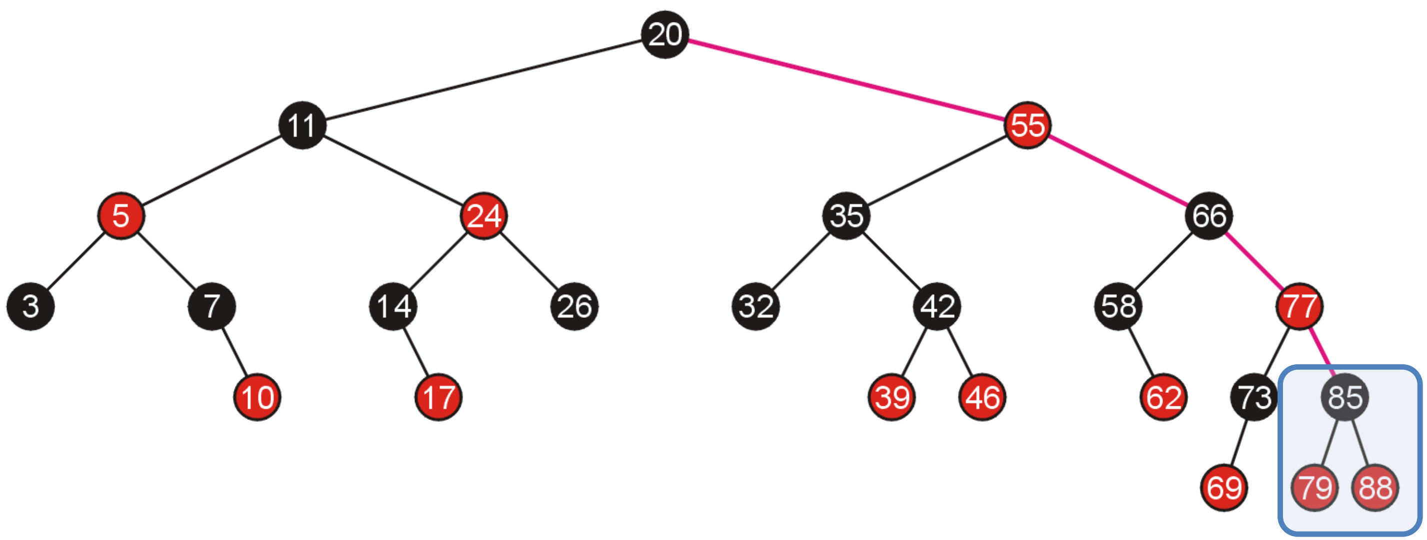
We continue and place 10 in the appropriate location

– No further rotations are required



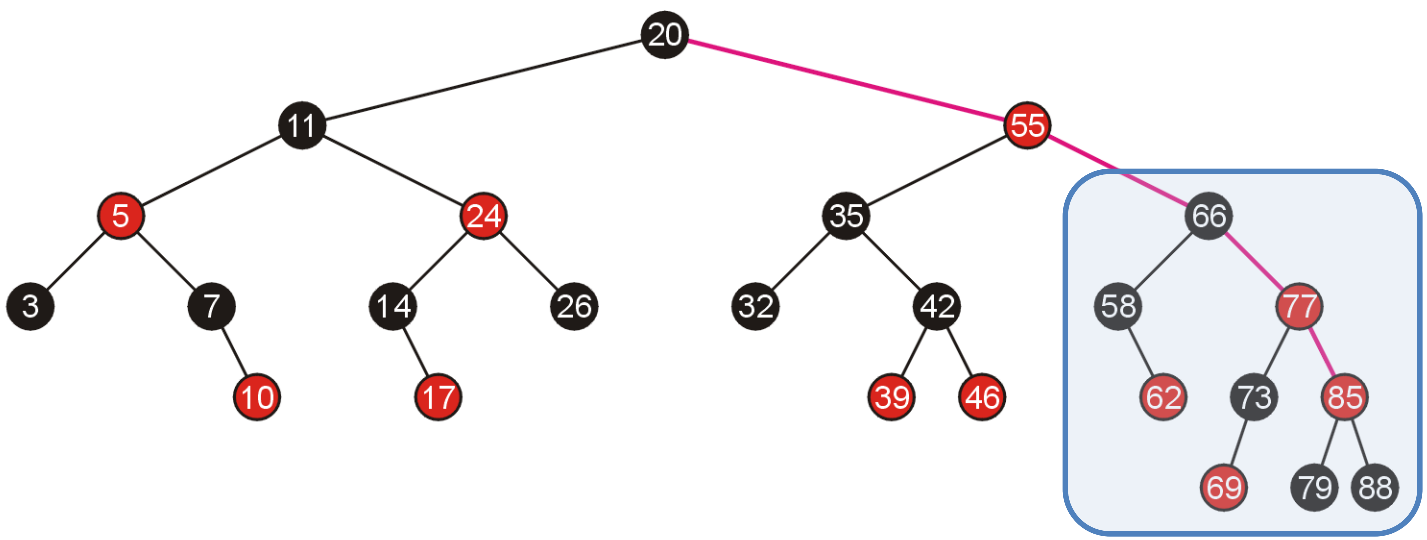
Examples of Top-Down Insertions: Insert 90

To add the node 90, we traverse down the right tree until we reach 85 which has two red children



Examples of Top-Down Insertions: Insert 90

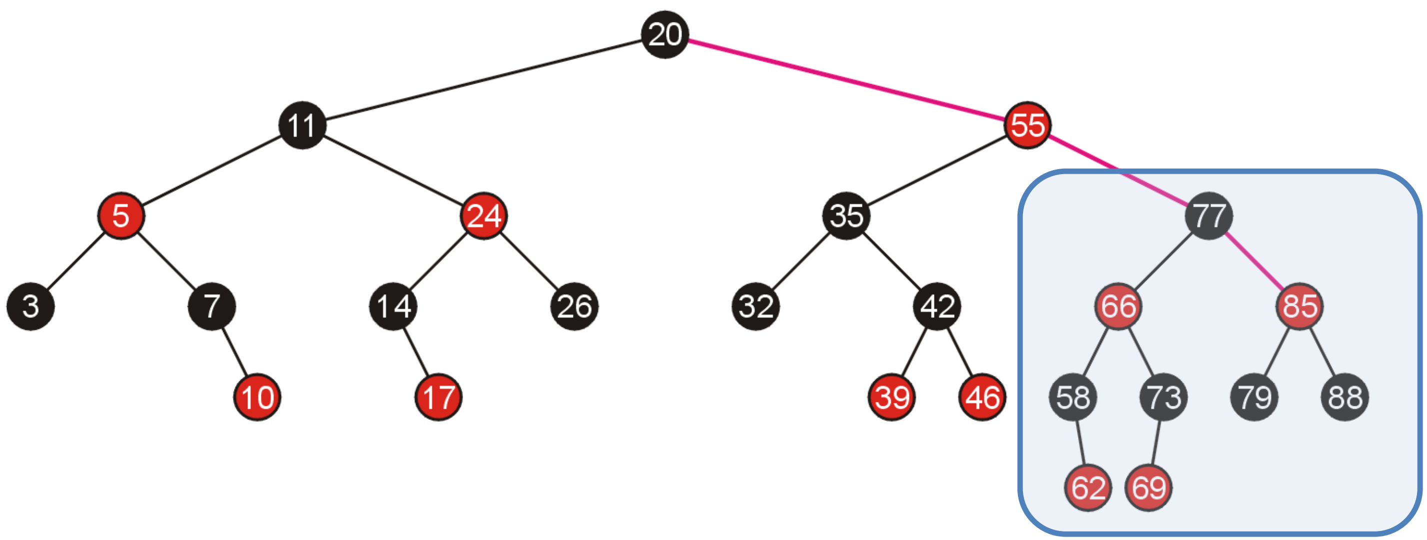
We **swap the colours**, however this creates a red-red pair between 85 and its parent



Examples of Top-Down Insertions: Insert 90

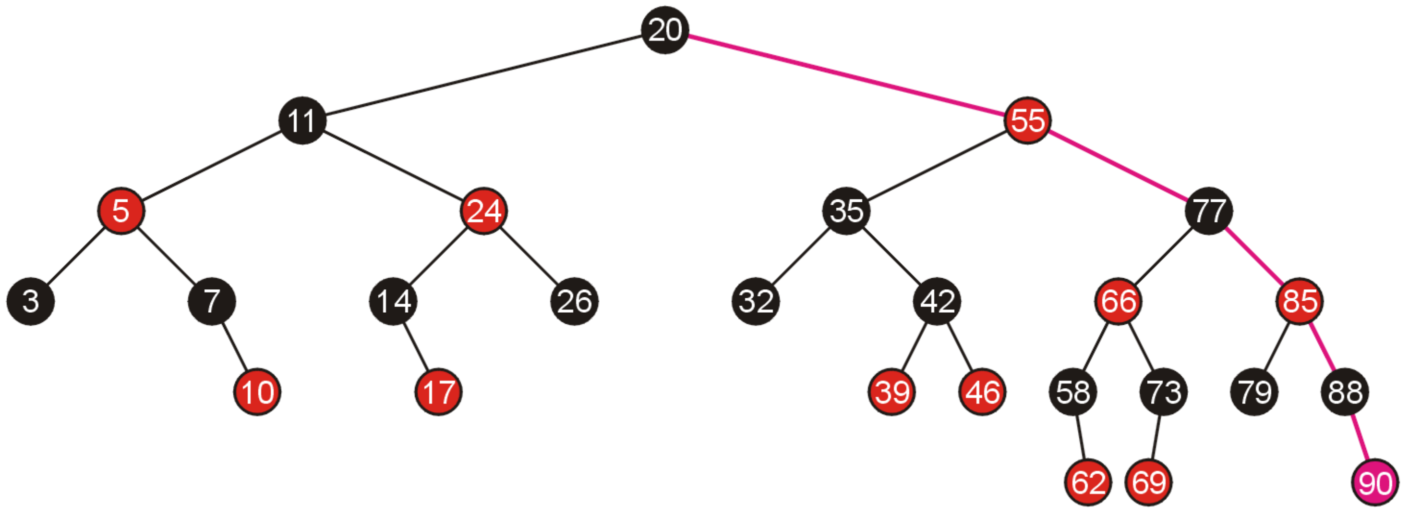
We require only one **rotation** to solve this problem.

This does not cause any problem for its parents.



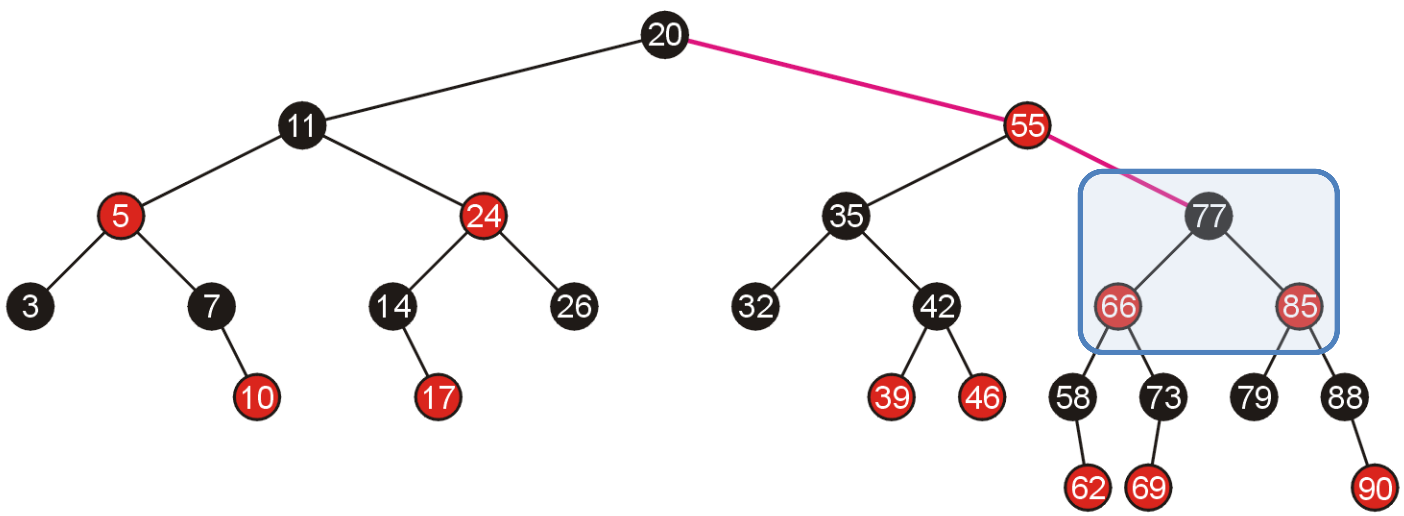
Examples of Top-Down Insertions: Insert 90

We continue to search down the right path and add 90 in the appropriate location—no further corrections are required



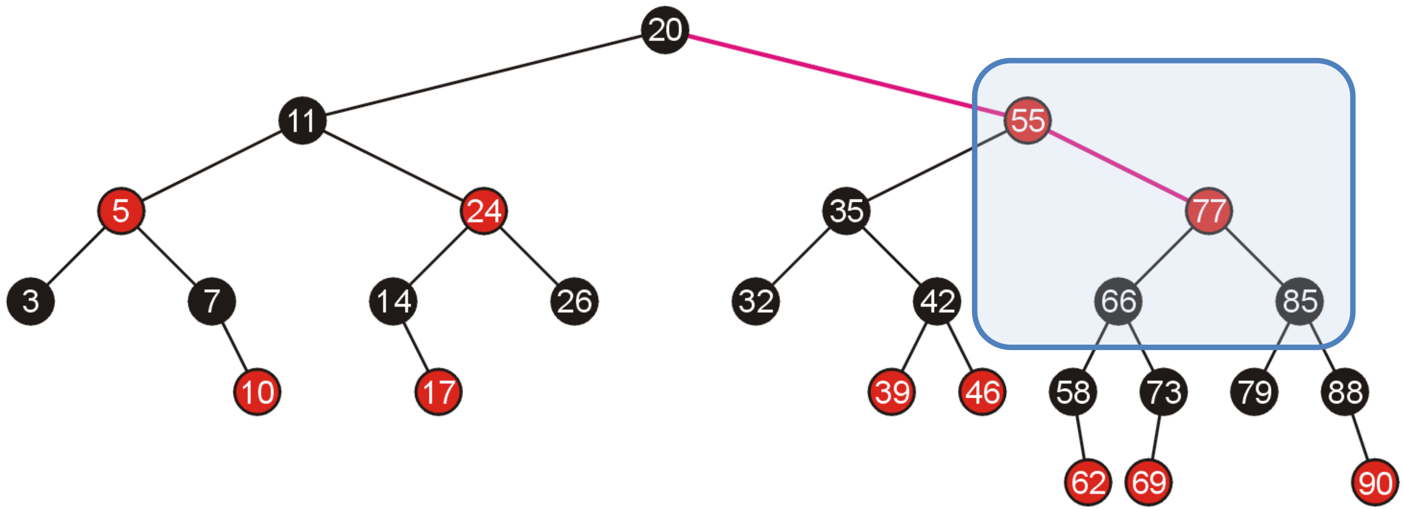
Examples of Top-Down Insertions: Insert 95

Next, adding 95, we traverse down the right-hand until we reach node 77 which has two red children



Examples of Top-Down Insertions: Insert 95

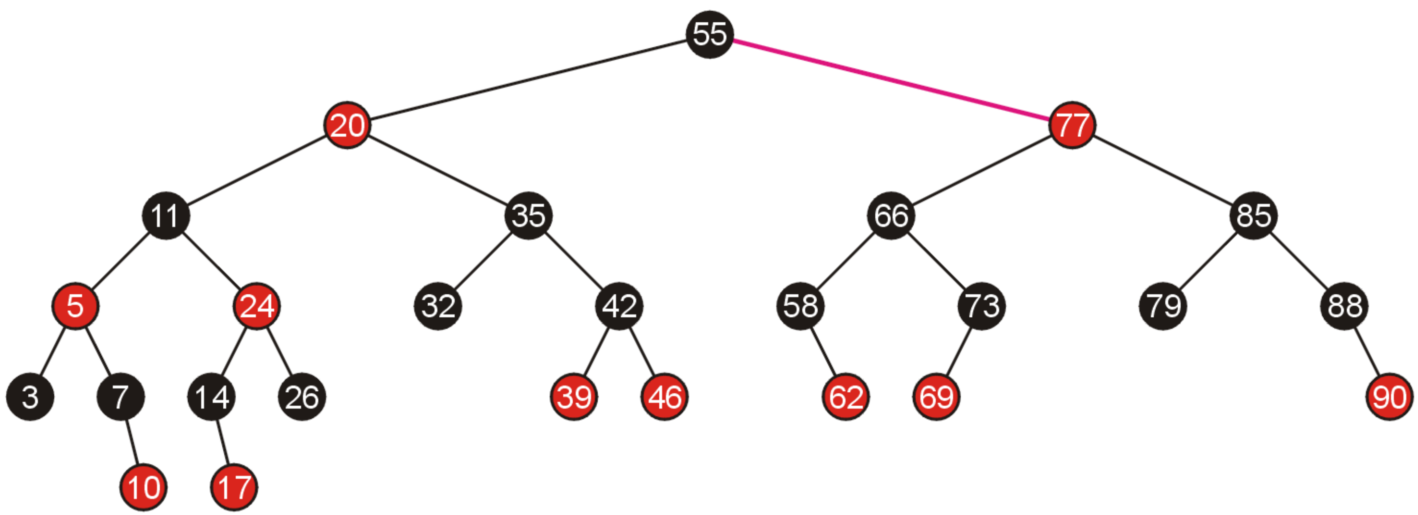
We **swap the colours**, which causes a red-red parent-child combination which must be fixed by a rotation



Examples of Top-Down Insertions: Insert 95

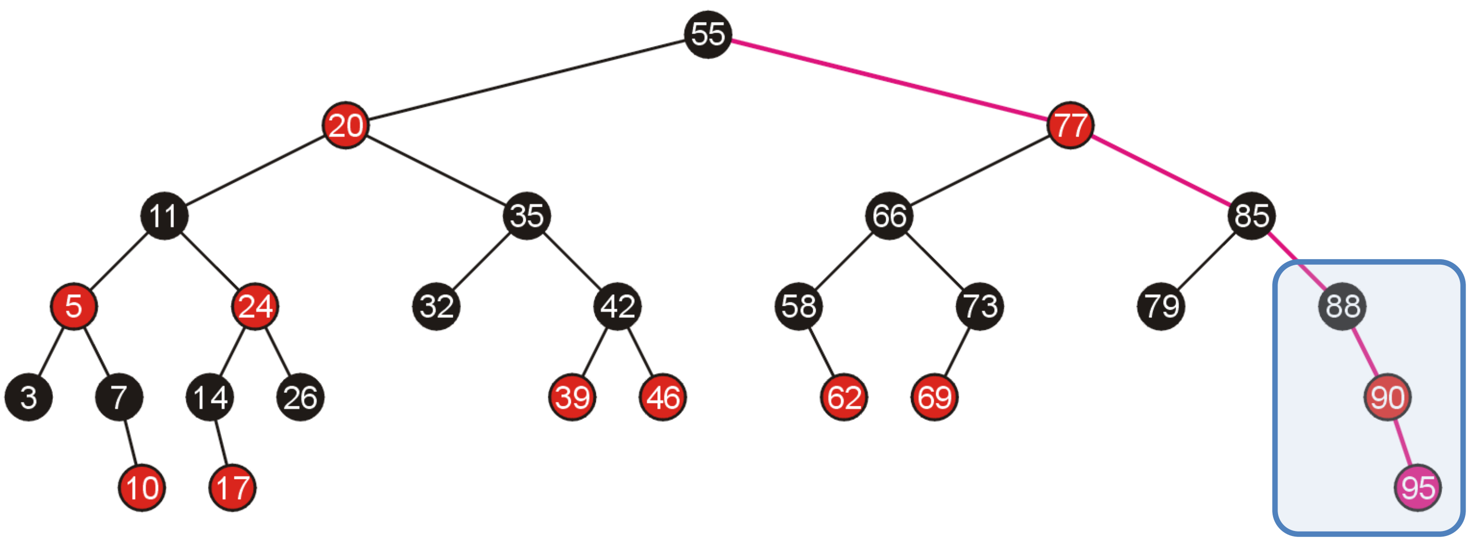
The **rotation** is around the root

- Note this rotation was not necessary with the bottom-up insertion of 95



Examples of Top-Down Insertions: Insert 95

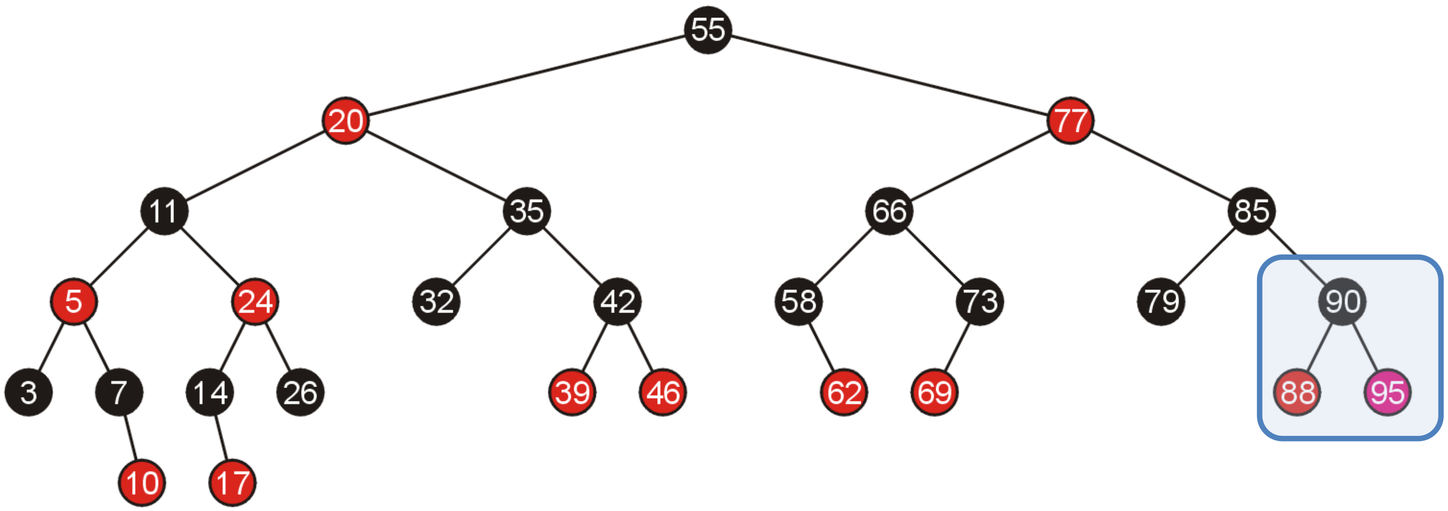
We can now proceed to add 95 by following the right-hand branch, and the insertion causes a red-red parent-child combination



Examples of Top-Down Insertions: Insert 95

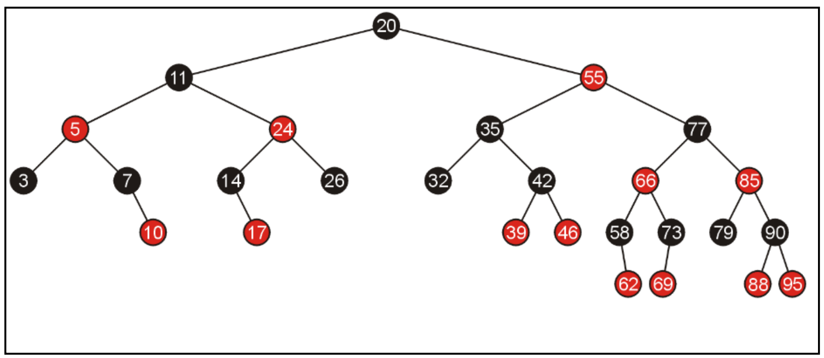
This is fixed with a single **rotation**

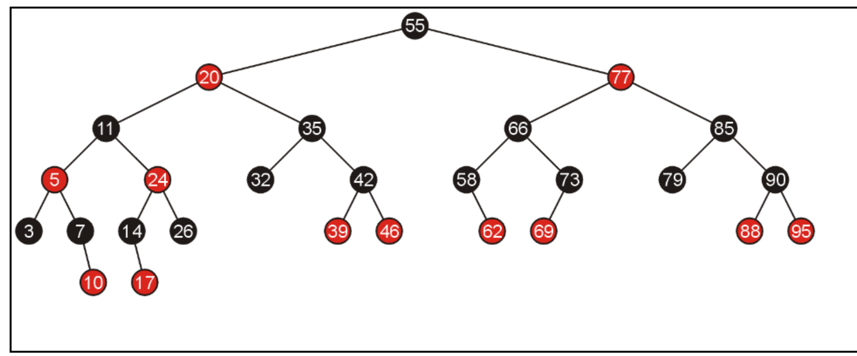
– We are guaranteed that this will not cause any further problems



Compare Top-Down and Bottom-up Insertions

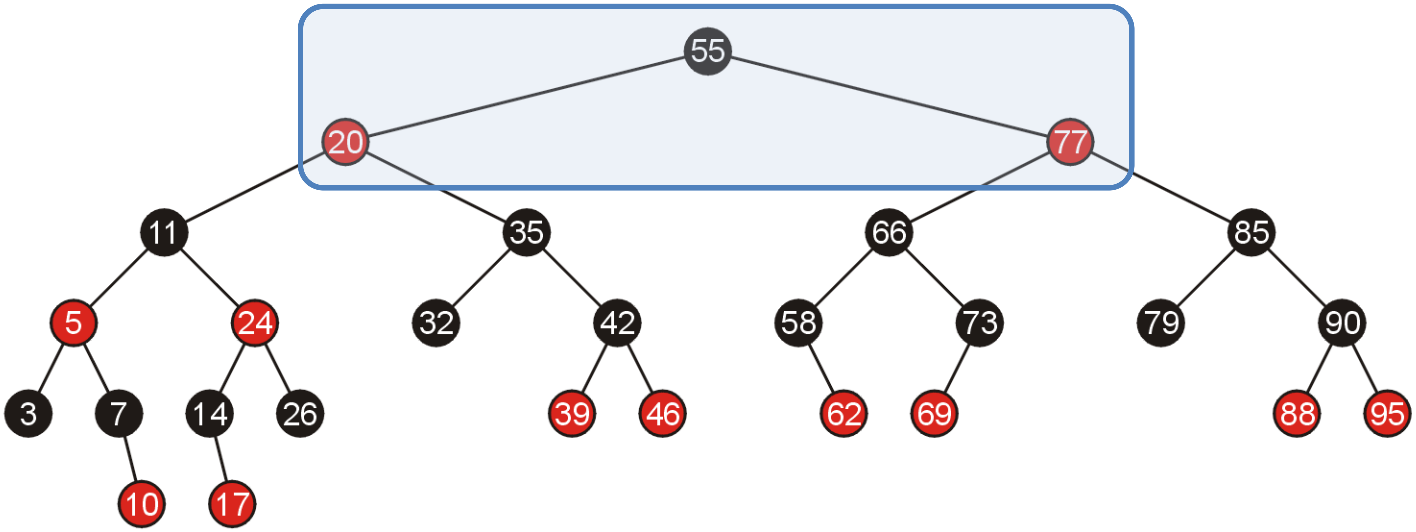
If we compare the result of doing bottom-up insertions (left, seen previously) and top-down insertions (right), we note the resulting trees are different, but both are still valid red-black trees





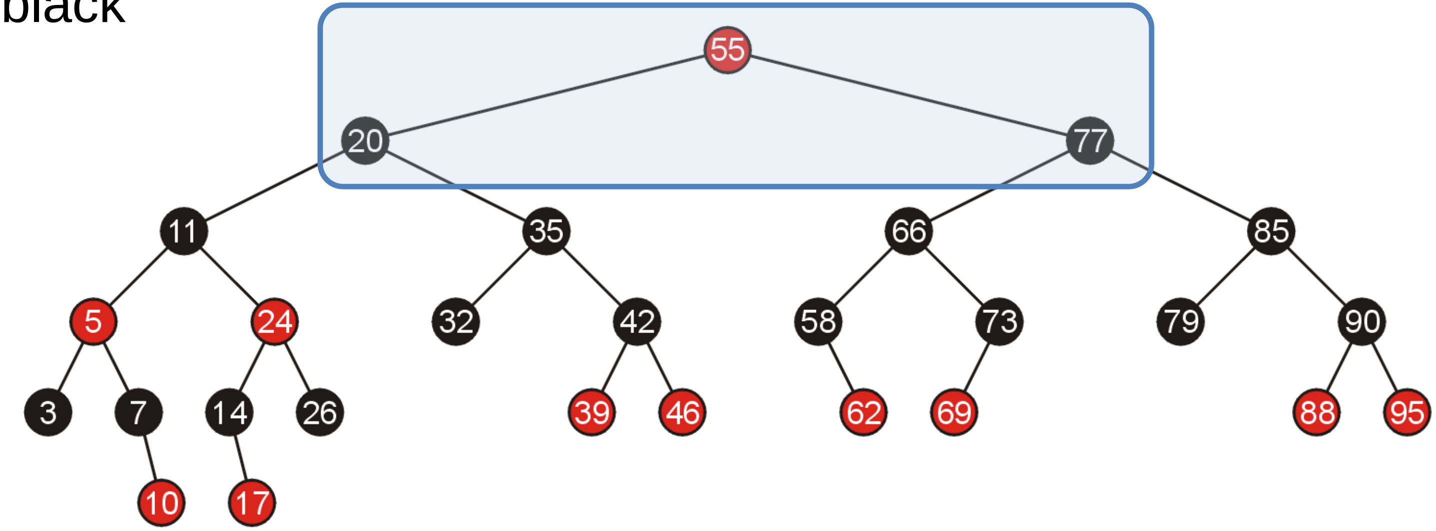
Examples of Top-Down Insertions: Insert 99

If we add 99, the first thing we note is that the root has two red children, and therefore we **swap the colours**



Examples of Top-Down Insertions: Insert 99

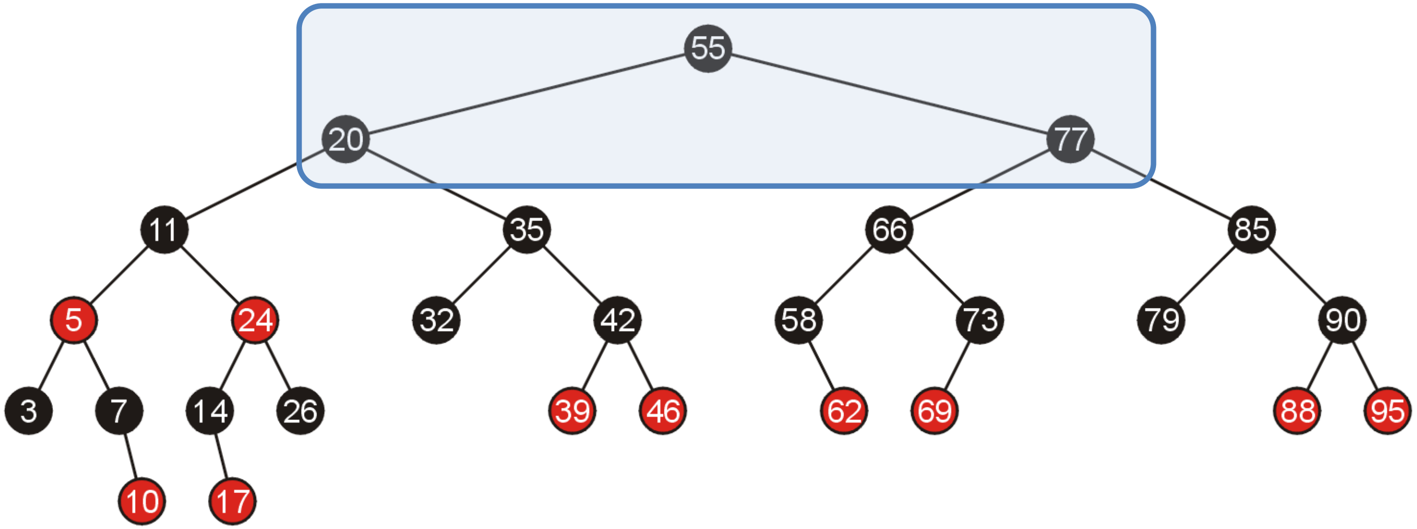
At this point, each path to a non-full node still has the same number of black nodes, however, we violate the requirement that the root is black



Examples of Top-Down Insertions: Insert 99

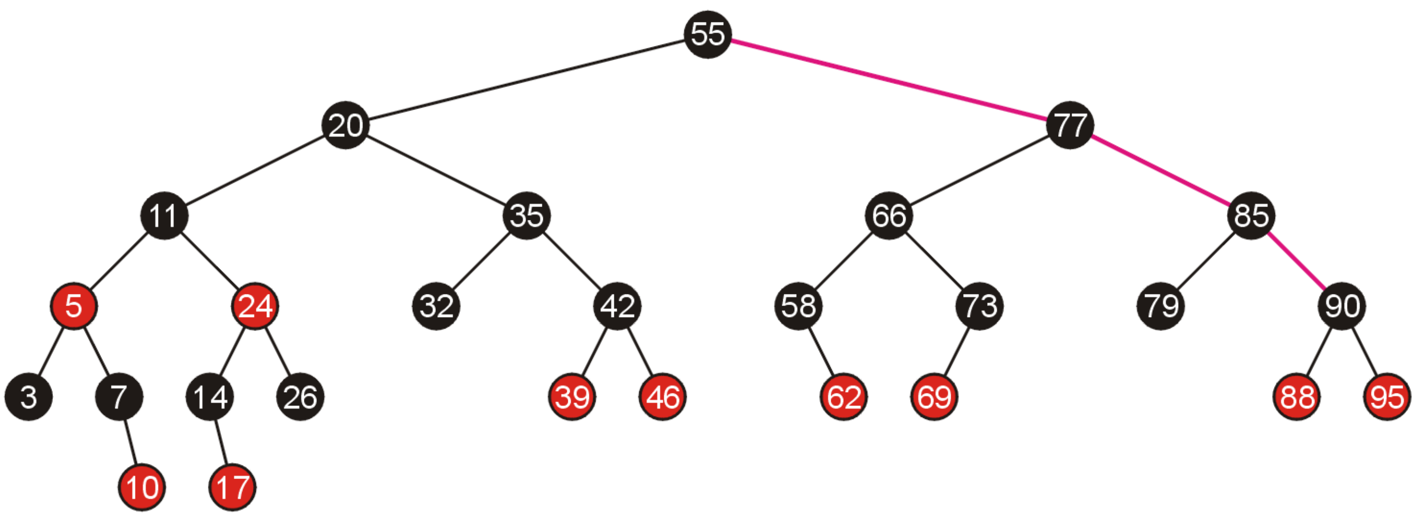
We **change the colour of the root to black**

– This adds one more black node to each path



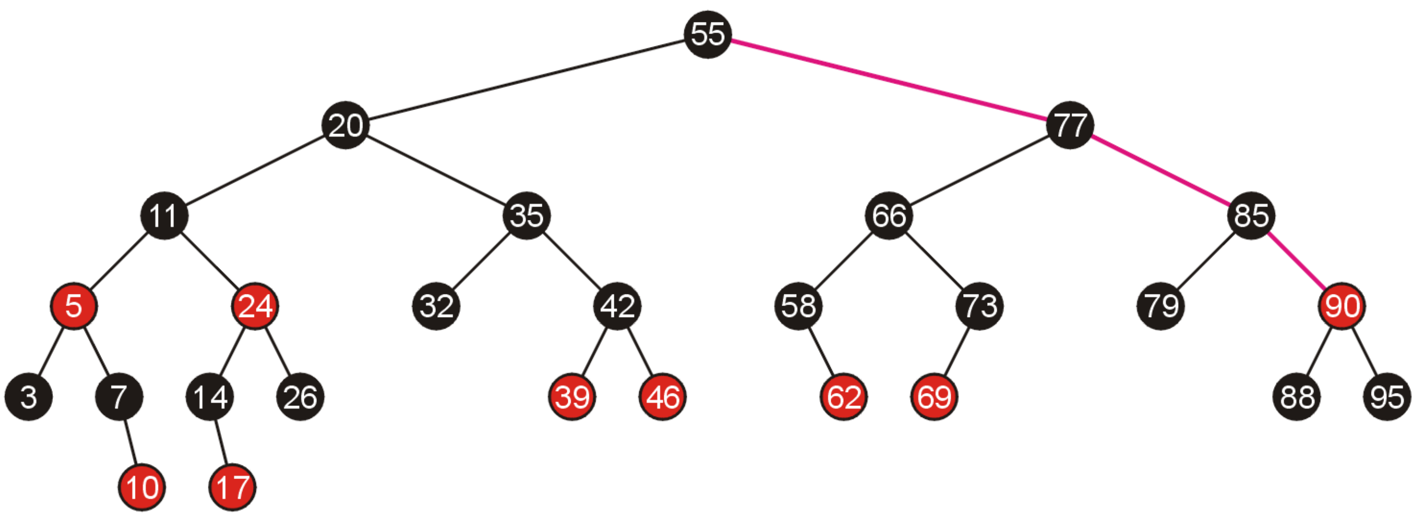
Examples of Top-Down Insertions: Insert 99

Moving to the right, we now reach node 90 which has two red children and therefore we **swap the colours**



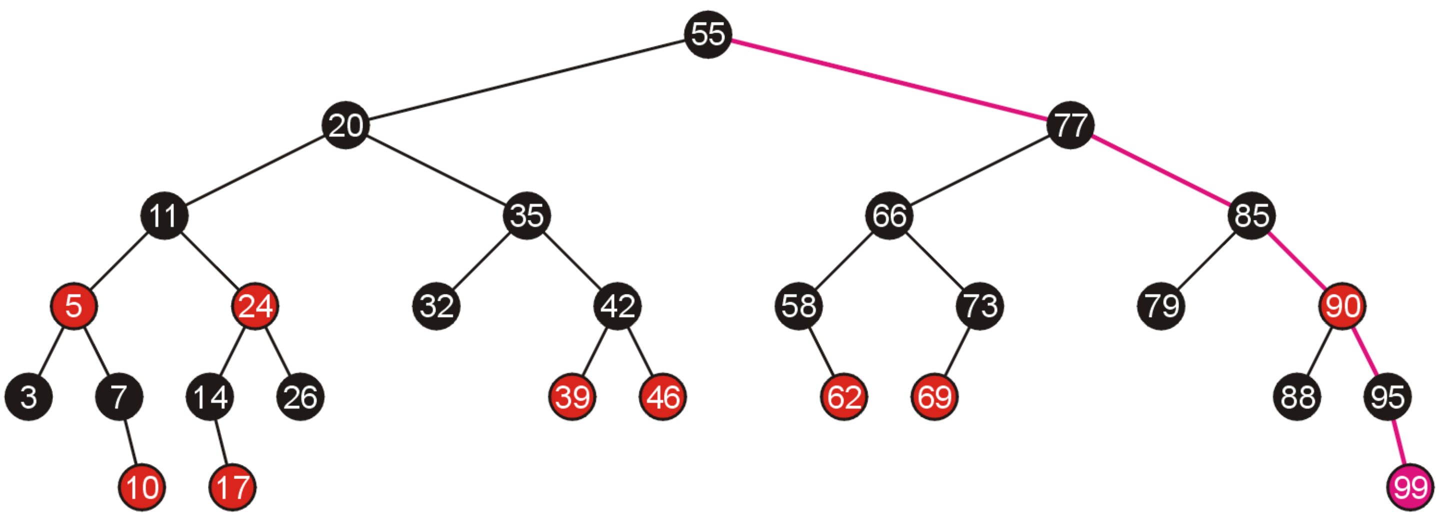
Examples of Top-Down Insertions: Insert 99

We continue down the right to add 99



Examples of Top-Down Insertions: Insert 99

This does not violate any of the rules of the red-black tree and therefore we are finished



Top-Down Deletions

- **Case #A: Deleting a red leaf node**

- **Case #B: Deleting a black leaf node**

  ➔ Complicated. Similar to Case #D3

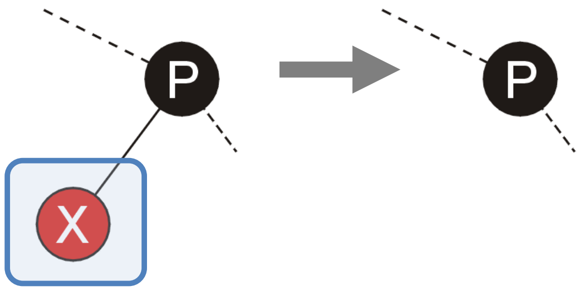
- **Case #C: Deleting a red/black node with one child**

- **Case #D: Deleting a full red/black node (i.e., with two children)**

  ➔ Complicated

Top-Down Deletions: Case #A

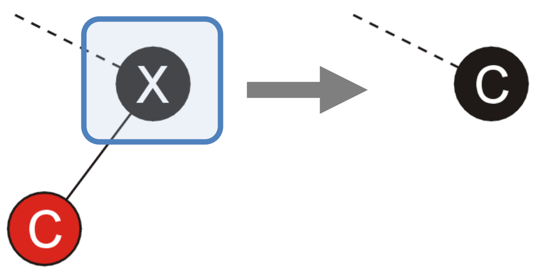
Case #A. If we are deleting a red leaf node X, then we are finished

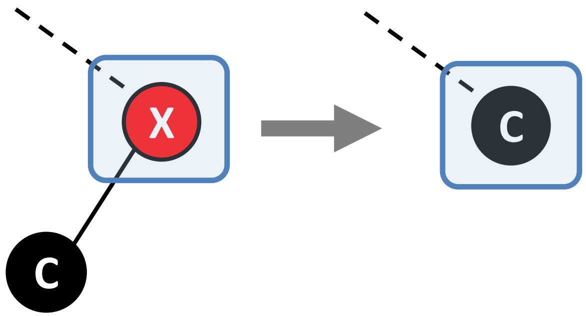


Top-Down Deletions: Case #C

Case #C. If we are deleting a node X with one child

➔ replace the value of X with the value of the leaf node





Top-Down Deletions: Case #D

Case #D. If we are deleting a full node, we use the same strategy used in standard binary search trees:

- 1) Locate the node “x” to be deleted
- 2) Locate the node “m”, which is the minimum element in the right sub-tree of “x”
- 3) Replace the node “x” with the node “m”
- 4) Delete the node “m”.

Top-Down Deletions: Case #D

That minimum node “m” must be either:

- **Case #D1**: “m” is a red leaf node,
- **Case #D2**: “m” is a black node with a single red leaf node, or
- **Case #D3**: “m” is a black leaf *node*

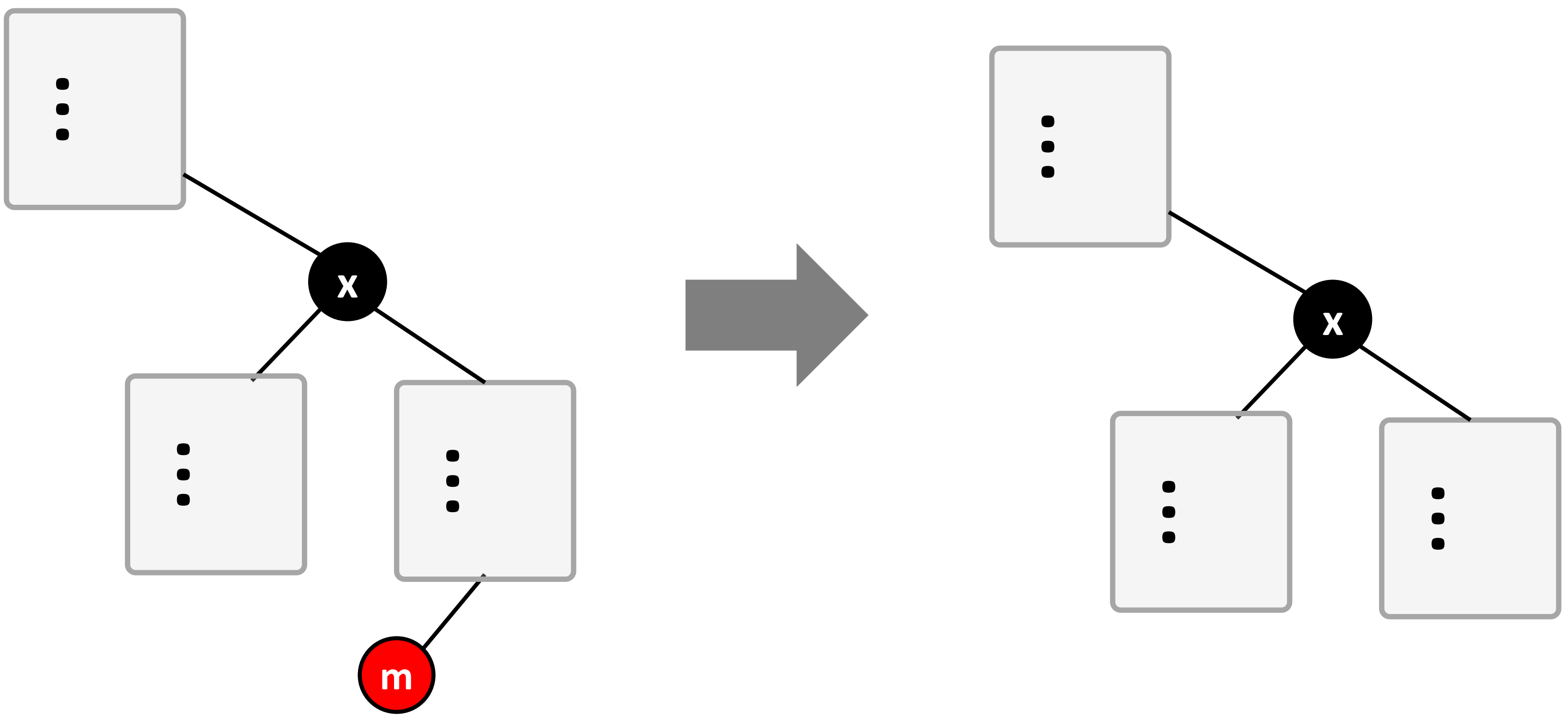
The first two cases are easy to solve.

For Case #D3, take the similar top-down insertion strategies.

See why RBTree is difficult? You should handle all different cases (nicely).

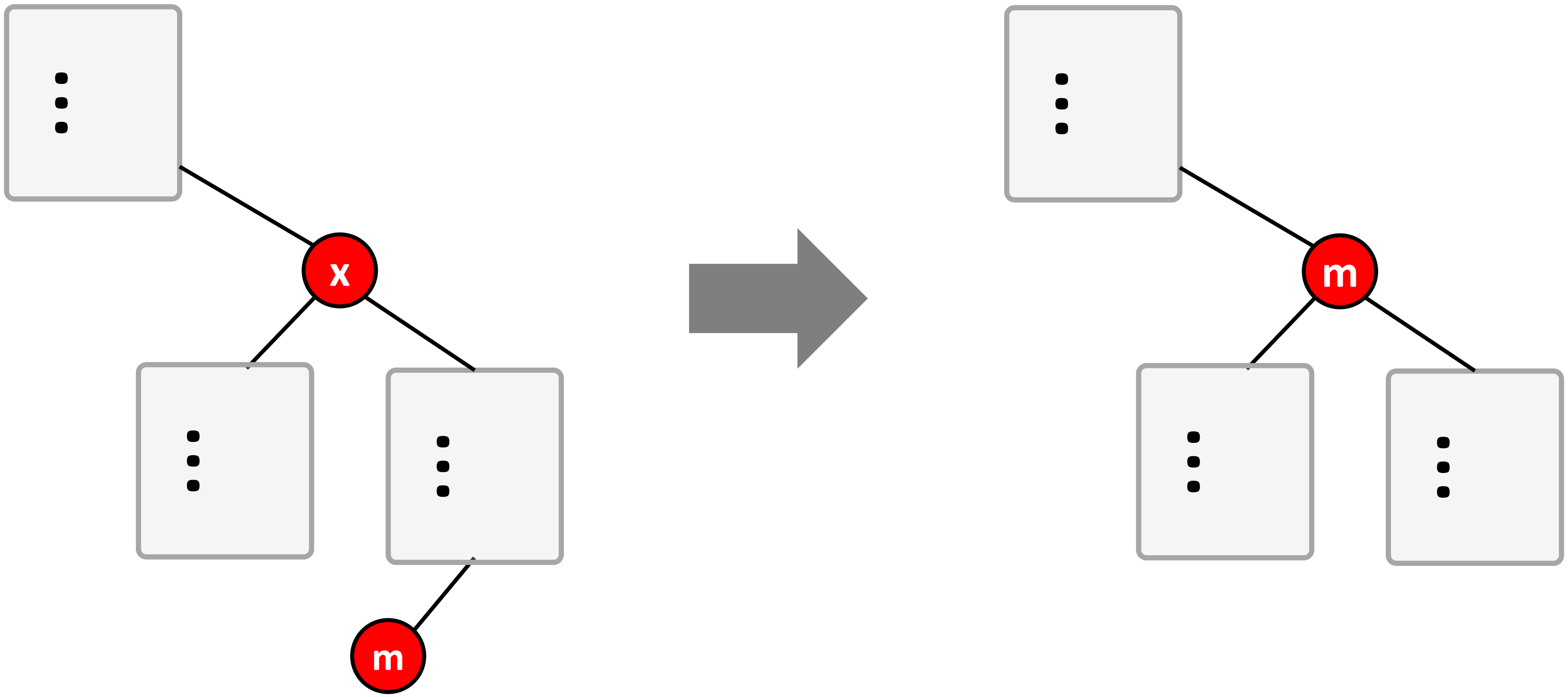
Top-Down Deletions: Case #D1

– Case #D1: “m” is a red leaf node ➔ Easy to solve
– Case #D1-A: “x” is black



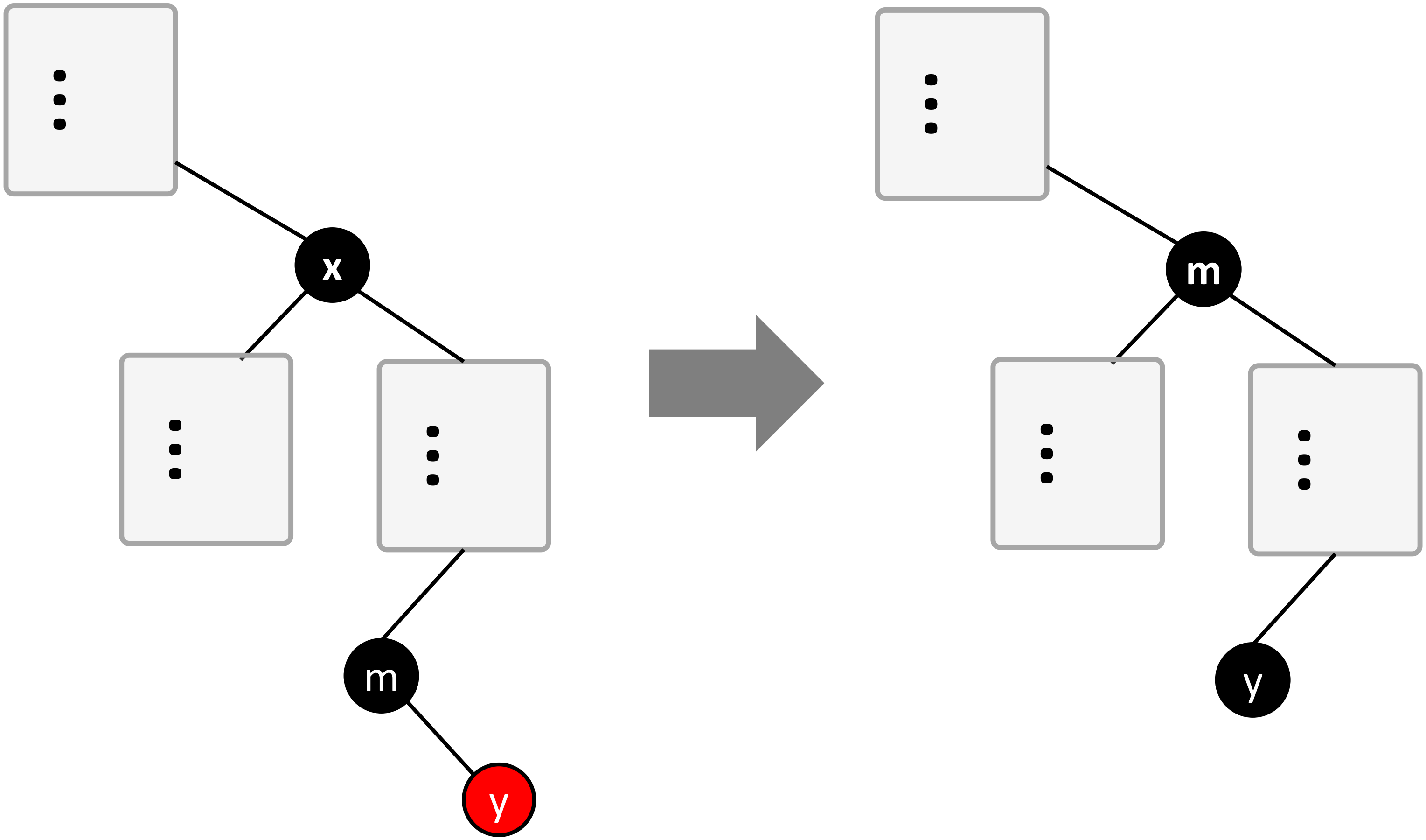
Top-Down Deletions: Case #D1

– Case #D1: “m” is a red leaf node ➔ Easy to solve
– Case #D1-B: “x” is red



Top-Down Deletions: Case #D2

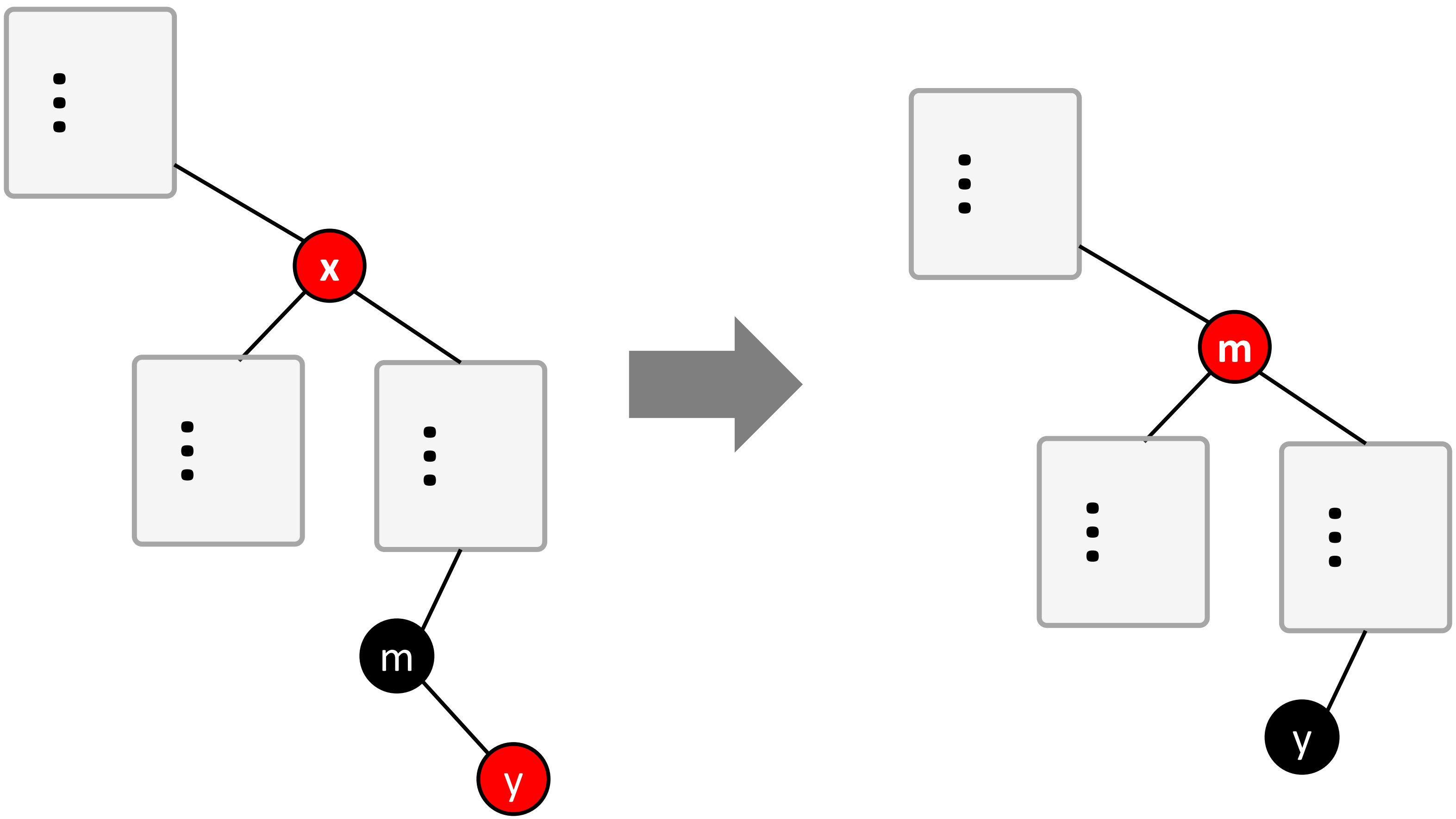
- Case #D2: “m” is a black node with a single red leaf node ➔ Easy to solve
- Case #D2-A: “x” is black



Top-Down Deletions: Case #D2

– Case #D2: “m” is a black node with a single red leaf node ➔ Easy to solve

– Case #D2-B: “x” is red

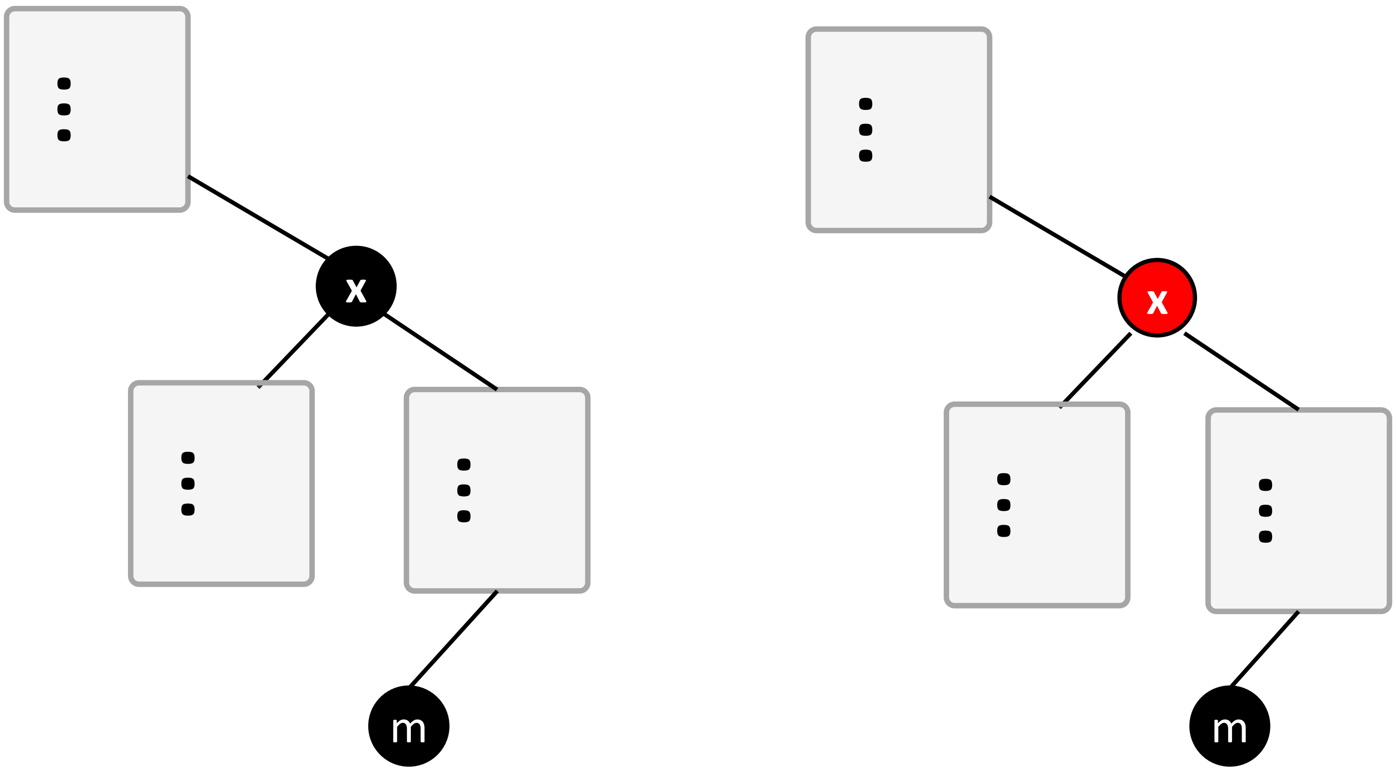


Top-Down Deletions: Case #D3

- Case #D3: “m” is a black leaf *node*

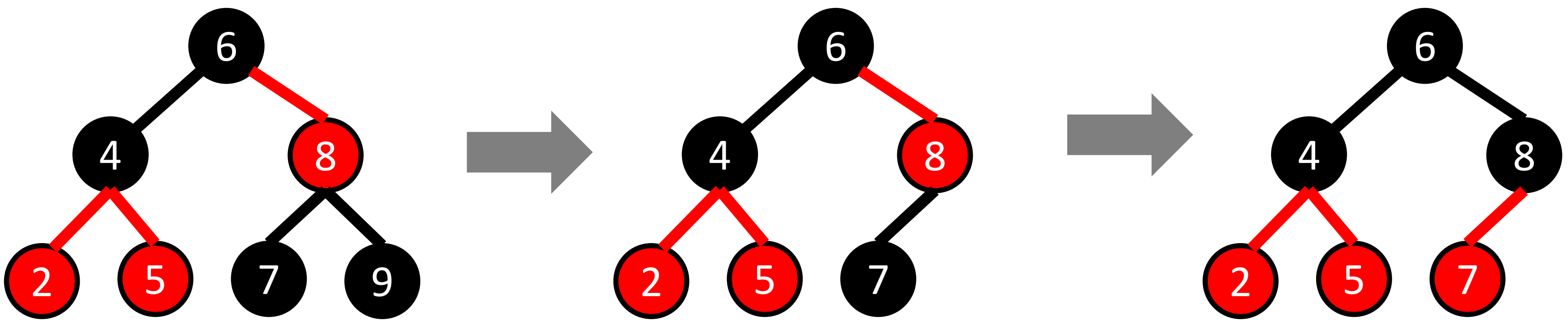
➔ take the similar top-down insertion strategies (rotate then recolour).

See why RBTree is difficult?
You should handle all different cases (nicely).



Top-Down Deletions: Examples

- **Delete 9**

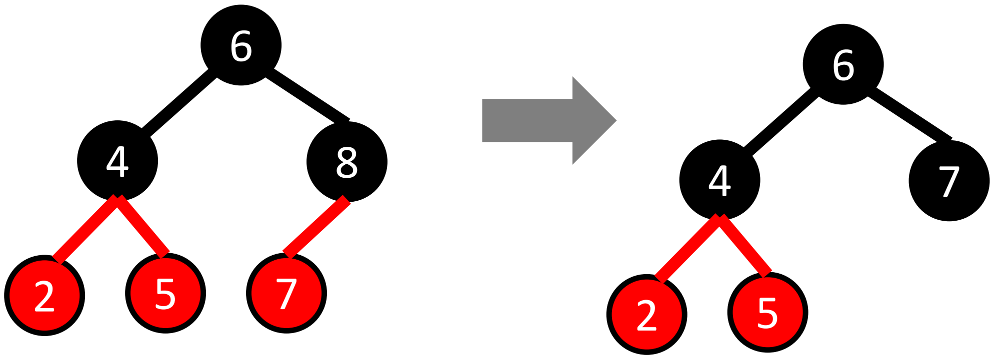


Remove 9,
but 8 becomes an issue

Swapping the color
solves the problem

Top-Down Deletions: Examples

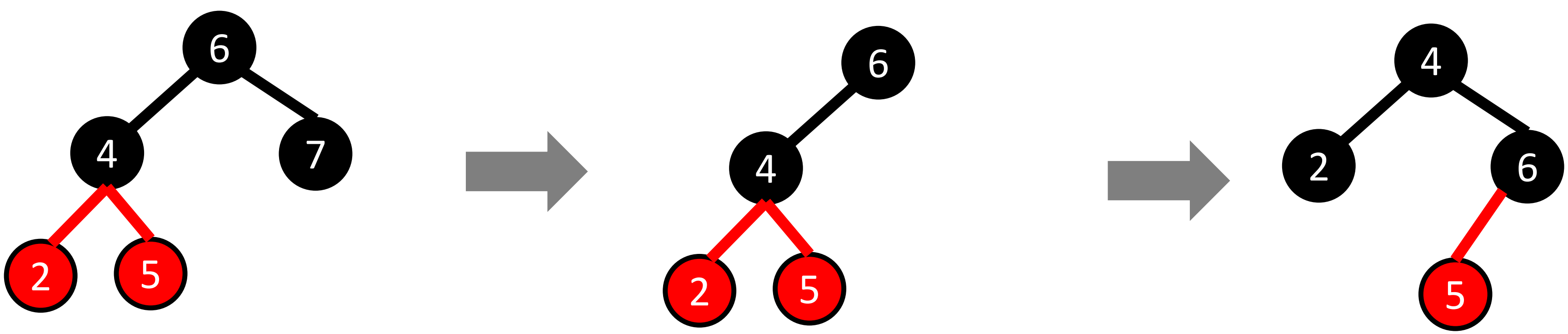
- **Delete 8**



Deleting 8 is an easy case

Top-Down Deletions: Examples

- **Delete 7**



Remove 7,
then 6 becomes unbalanced

Rotate and recolor
solves the problem

Maximum height of Red-Black Trees

- **Theorem:**
  - RBTree with n internal nodes has a maximum height, 2 log(n+1).

- **Proof sketch:**
  - Step 1. Proof that RBTree has at least $2^{bh(x)}$-1 internal nodes
    - where "x" is a root node and bh(x) is the maximum number of black nodes on any path from x
  - Step 2. Then leverage the definition of RBTree (i.e., null-path balance)

- **Want you to proof this on your own. Will be discussing the full proof later.**

AVL Vs. Red-Black Trees

	Average	Worst-case
Space	O(n)	O(n)
Lookup	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

AVL tree

	Average	Worst-case
Space	O(n)	O(n)
Lookup	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Red-Black Tree

Asymtotic complexity
for lookup/insert/delete is the same!

AVL Vs. Red-Black Trees

- AVL VS RBTree
  - AVL maintains its balance more tight than RBTree
    - Recall the definition
  - AVL performs better for lookup-intensive applications
  - RBTree provides faster worst-case performance for insert/delete

```
To quote Linux Weekly News:

    There are a number of red-black trees in use in the kernel.
    The deadline and CFQ I/O schedulers employ rbtrees to
    track requests; the packet CD/DVD driver does the same.
    The high-resolution timer code uses an rbtree to organize outstanding
    timer requests.  The ext3 filesystem tracks directory entries in a
    red-black tree.  Virtual memory areas (VMAs) are tracked with red-black
    trees, as are epoll file descriptors, cryptographic keys, and network
    packets in the "hierarchical token bucket" scheduler.
```

Red-Black Trees

In this topic, we have covered red-black trees

- simple rules govern how nodes must be distributed based on giving each node a colour of either red or black
- insertions and deletions may be performed without recursing back to the root
- only one bit is required for the “colour”
- this makes them, under some circumstances, more suited than AVL trees

References

Wikipedia, http://en.wikipedia.org/wiki/Hash_function

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

[2] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.

[3] Robert Sedgewick, Left-Leaning Red-Black Trees, https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf