CS101 Algorithms and Data Structures

Red-Black Trees

Textbook Ch 13

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

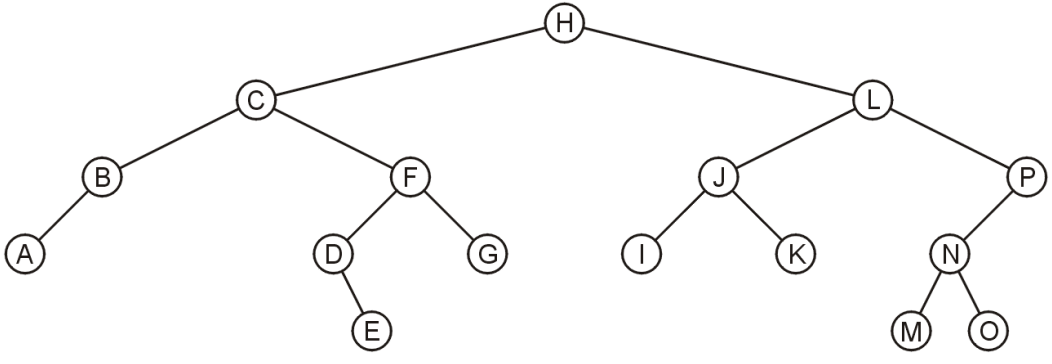
Red-Black Trees

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:



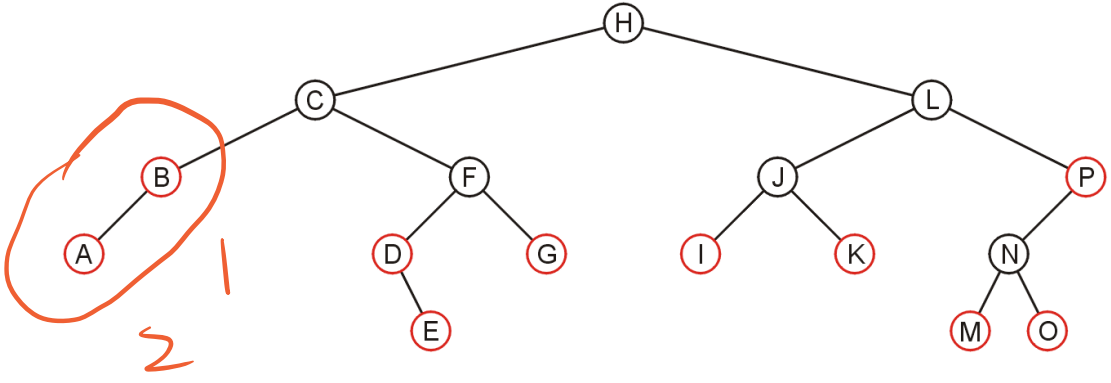
Red-Black Trees

All null paths include:

(H, C, **B**) (H, C, F, **D**) (H, L, J, **I**) (H, L, **P**)

(H, C, B, **A**) (H, C, F, D, **E**) (H, L, J, **K**) (H, L, P, N, **M**)

(H, C, F, **G**) (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可不同

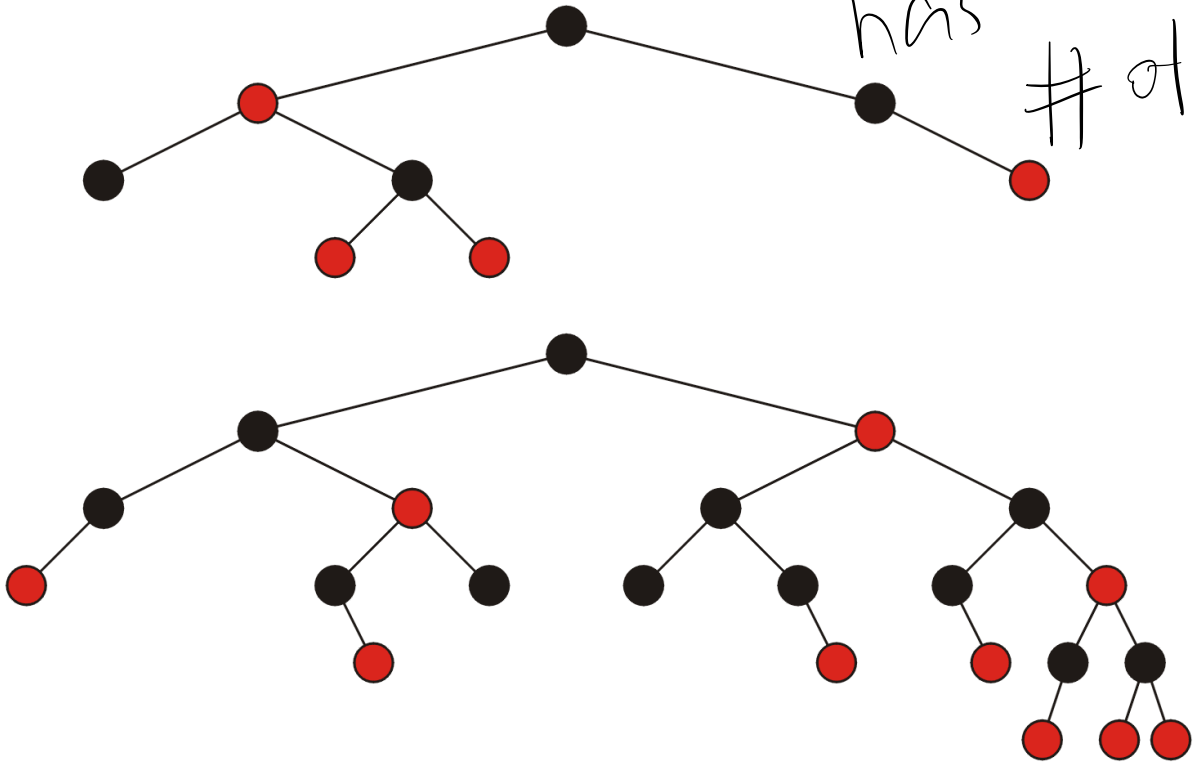
1、root B

2. red parent, black

4. any path start from root end at

Red-Black Trees

These are two examples of red-black trees:



Red - full / leaf

Red-Black Trees

Theorem:

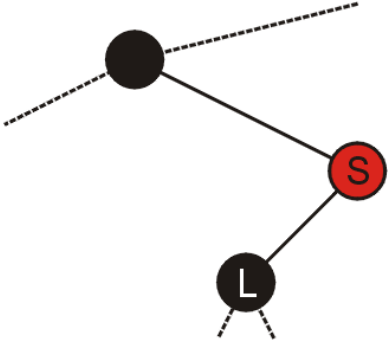
Every red node must be either

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

Proof:

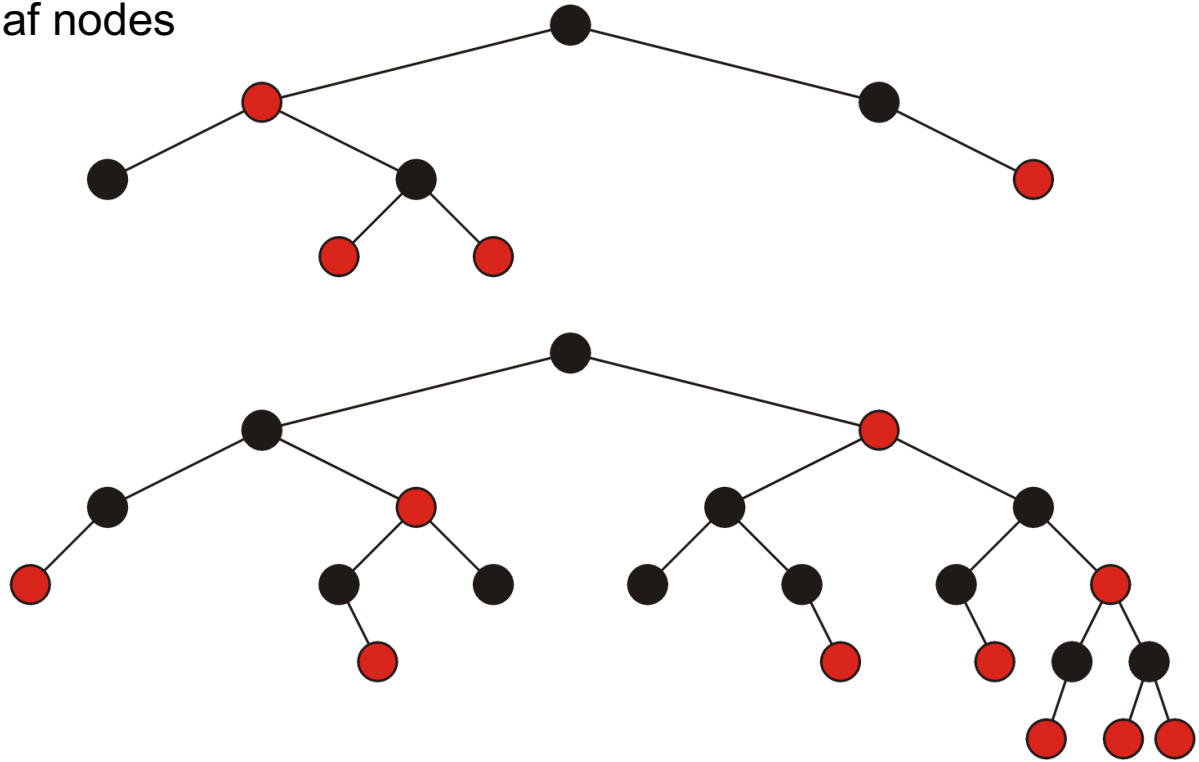
Suppose node S has one child:

- 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



1. Red leaf

Red-Black Trees

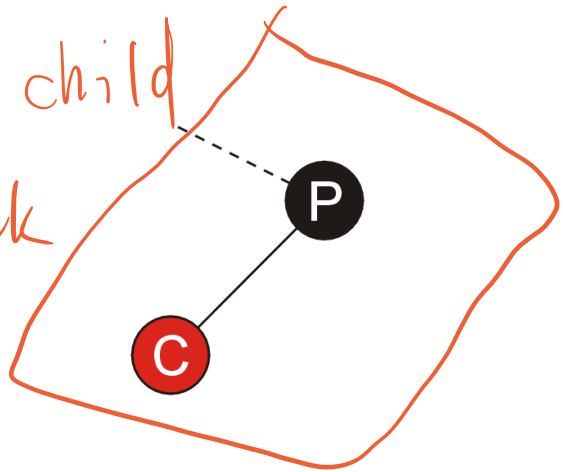
Another consequence is that if a node P has exactly one child:

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

leaf A has 1 child

1. Red leaf child

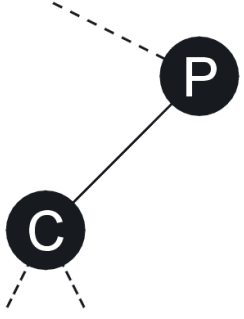
2. A black



Red-Black Trees

Suppose that the child is black

- then the null path ending in P will have one fewer black nodes than the null path passing through C
- this contradicts the requirement that each null path has the same number of black nodes
- therefore the one child must be red

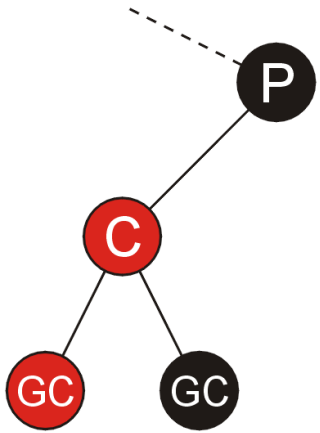


Red-Black Trees

Suppose the child is not a leaf node:

– Since it is red, its children:
  - cannot be red because that would contradict the requirement that all children of a red node are black, and
  - cannot be black, as that would cause any leaf node under the child to have more black nodes than the null path ending in P

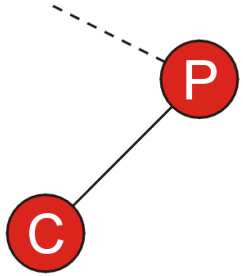
– Contradiction, therefore the child must be a leaf node



Red-Black Trees

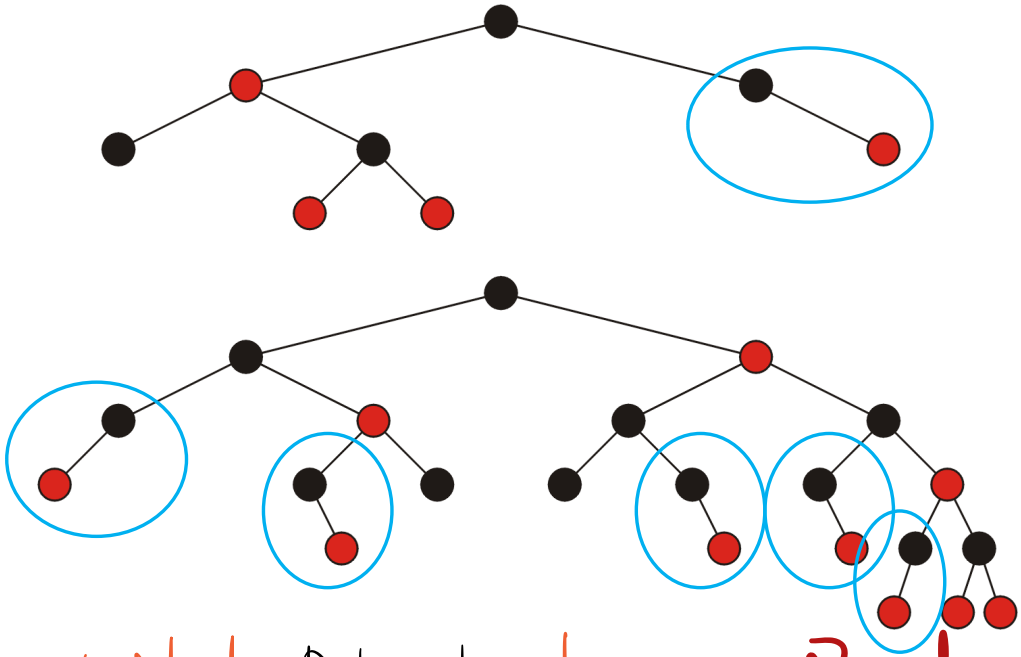
Since the one child is red and it must be a leaf node, the parent P must be black

- otherwise, we would contradict the requirement that the child of a red node must be black



Red-Black Trees

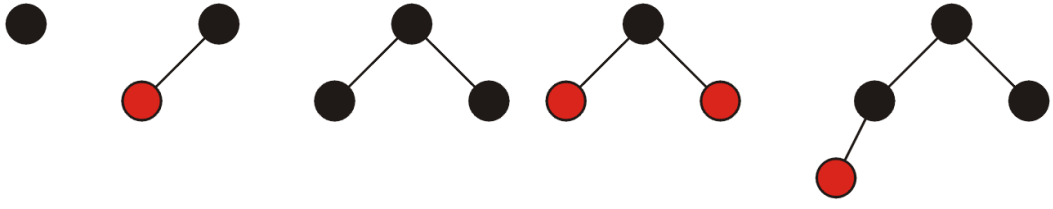
Again, in our examples, the only nodes with a single child are black and the corresponding children are red



A. 1 child Black have Red leaf

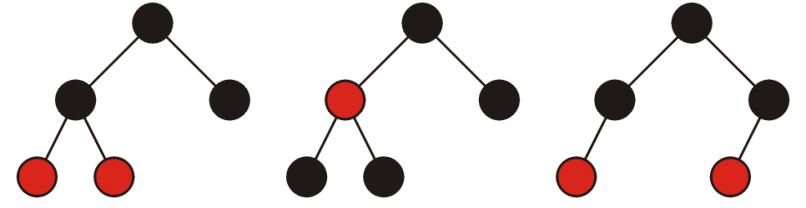
Red-Black Trees

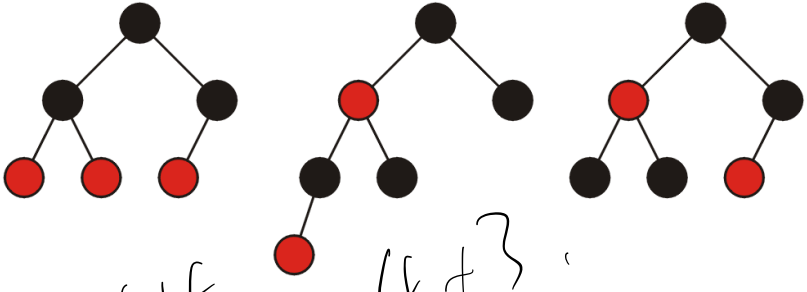
All red-black trees with 1, 2, 3, and 4 nodes:



Red-Black Trees

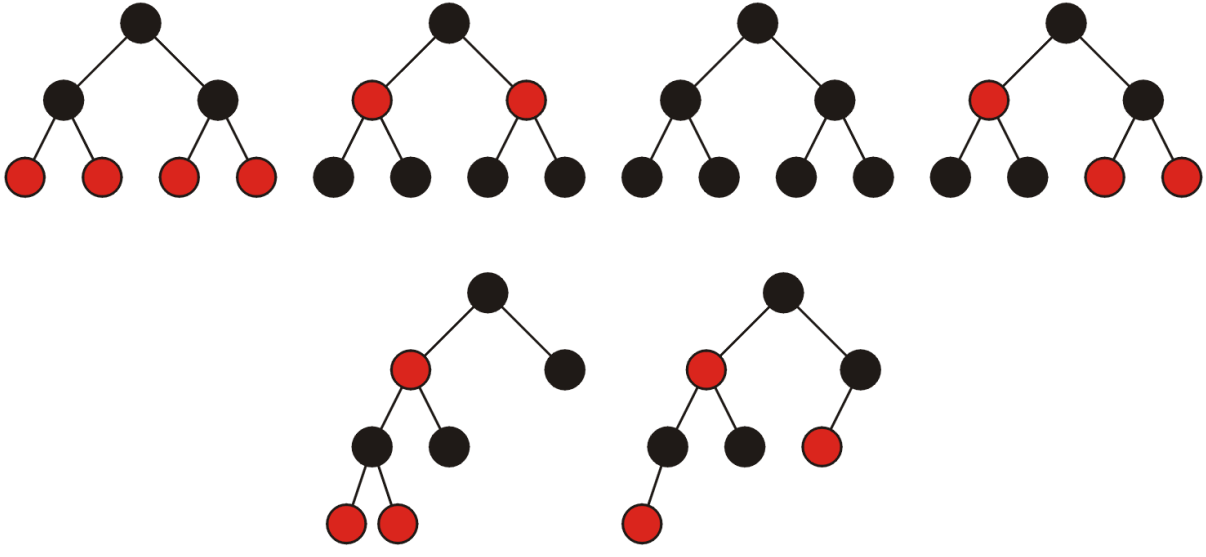
All red-black trees with 5 and 6 nodes:





Red-Black Trees

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



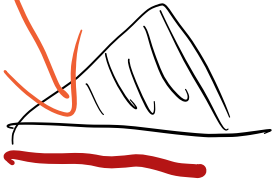
Red-Black Trees

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

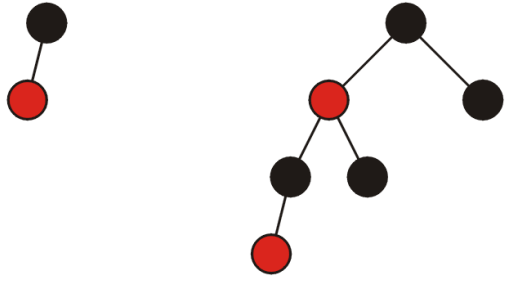
What is the worst case?



Red-Black Trees

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

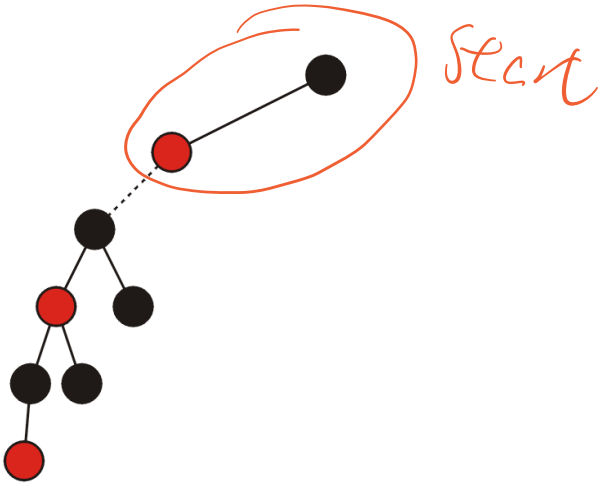
The following are the worst-case red-black trees with 1 and 2 black nodes per null path (*i.e.*, heights 1 and 3)



ex: worst case of N black per path

Red-Black Trees

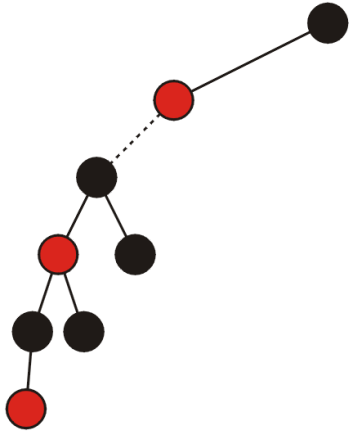
To create the worst-case for paths with 3 black nodes per path, start with a black and red node and add the previous worst-case for paths with 2 nodes



Red-Black Trees

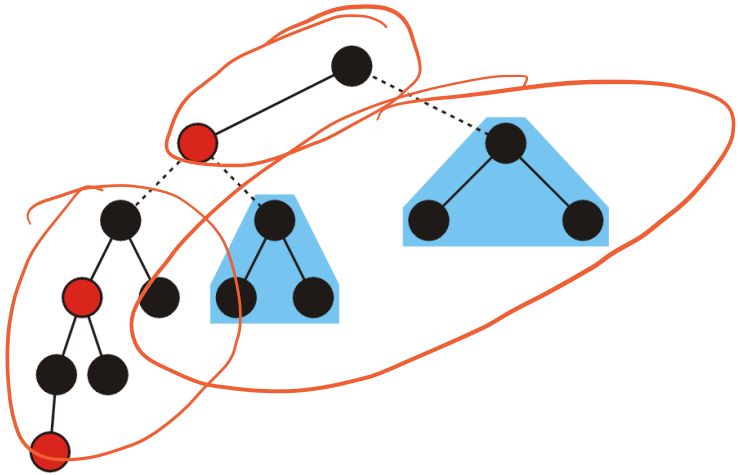
This, however, is not a red-black tree because the two top nodes do not have paths with three black nodes

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



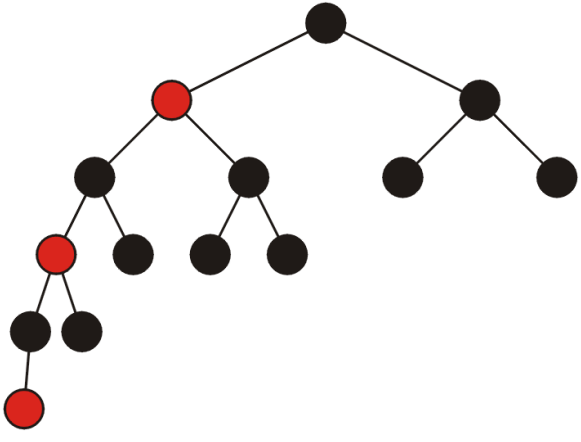
Red-Black Trees

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



Red-Black Trees

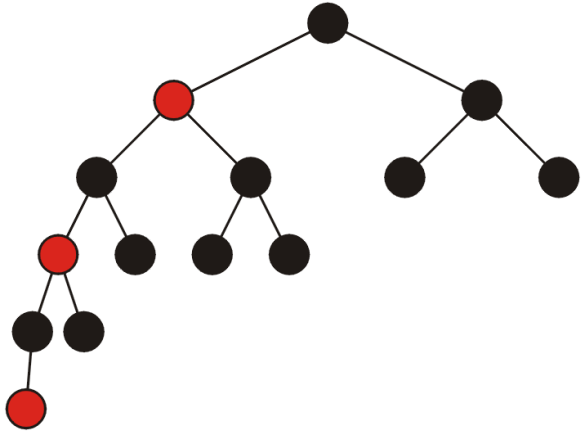
Thus, we have the worst-case for a red-black tree with three black nodes per path (or a red-black tree of height 5)



Red-Black Trees

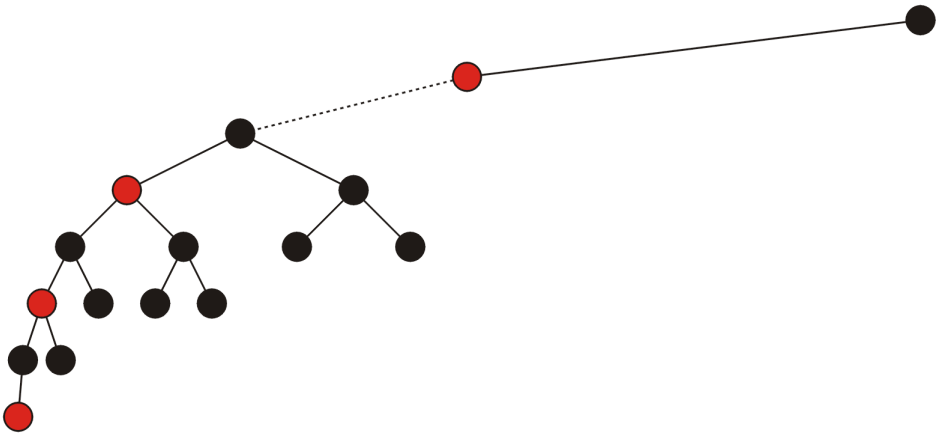
Note that the left sub-tree of the root has height 4 while the right has height 1

- Thus suggests that AVL trees may be better...



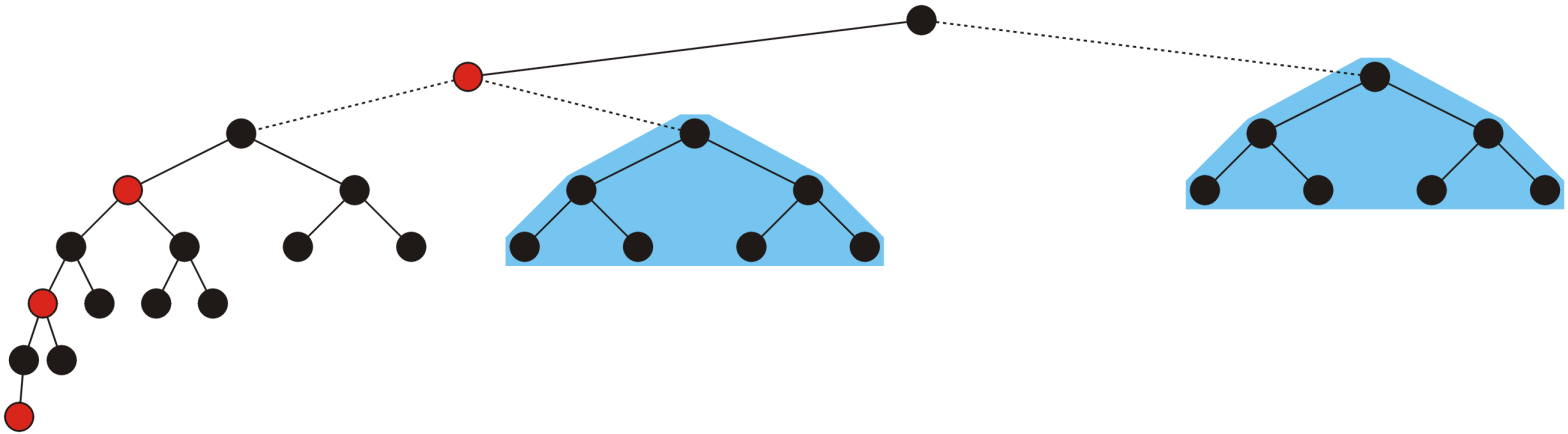
Red-Black Trees

To create the worst-case scenario for red-black trees with 4 black nodes per null path (*i.e.*, height 7), again, start with two nodes and add the previous worst-case tree:



Red-Black Trees

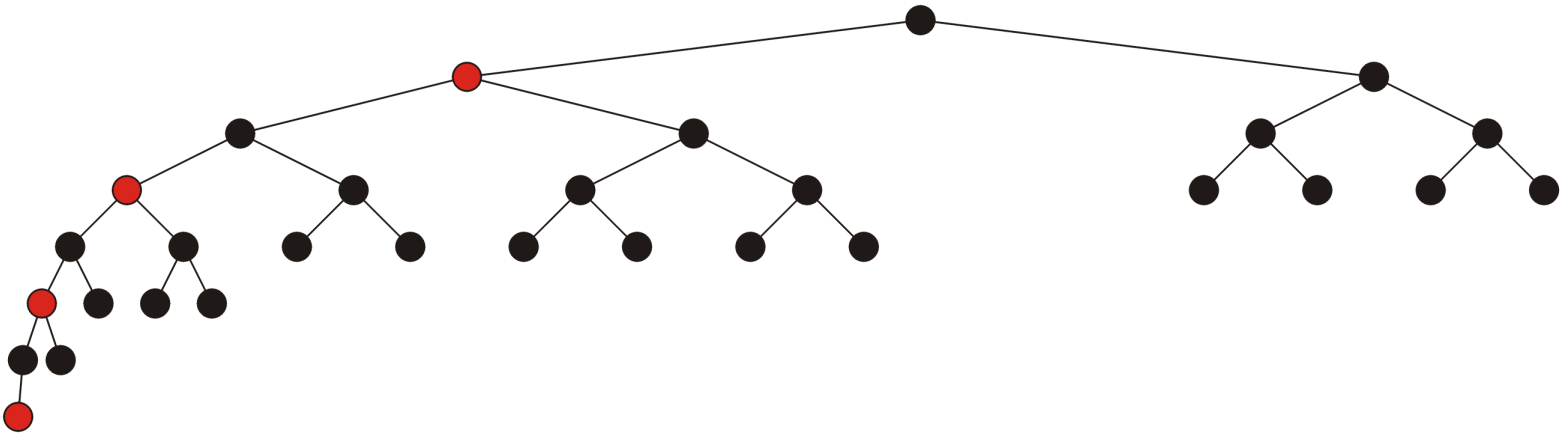
To satisfy the definition of the red-black tree, add two perfect trees of height 2:



Red-Black Trees

Thus, with 30 nodes, the worst case red-black tree has height 7

– The worst-case AVL tree with 33 nodes has height 6...



Red-Black Trees

Let k represent the number of black nodes per null path and $h = 2k - 1$ represent the height of the worst-case tree

To determine $F(k)$, the number of nodes in the worst-case tree, we note that $F(1) = 2$ and:

$$F(k) = F(k-1) + 2 + 2(2^{k-1} - 1)$$

the number of nodes in the worst-case red-black tree with $k - 1$ black nodes per null path

the two extra nodes

two perfect trees of height $k - 2$

Red-Black Trees

Use Maple:

```
> rsolve( {F(1) = 2, F(k) = 2 + F(k-1) + 2*(2^(k–1)–1)}, F(k) );
```

$$2\,2^{k}-2$$

```
> solve( {h = 2*k - 1}, k );    # solve equation for k
```

$$\left\{k=\frac{h}{2}+\frac{1}{2}\right\}$$

```
> eval( 2*2^k - 2, % );          # evaluate at k = h/2 – 1/2
```

$$2\,2^{\left(\frac{h}{2}+\frac{1}{2}\right)}-2$$

```
> solve( {n = %}, h );           # solve for h
```

$$\left\{h=\frac{2\ln\left(\frac{n}{2}+1\right)-\ln(2)}{\ln(2)}\right\}$$

Red-Black Trees

A little manipulation shows that the worst-case height simplifies to

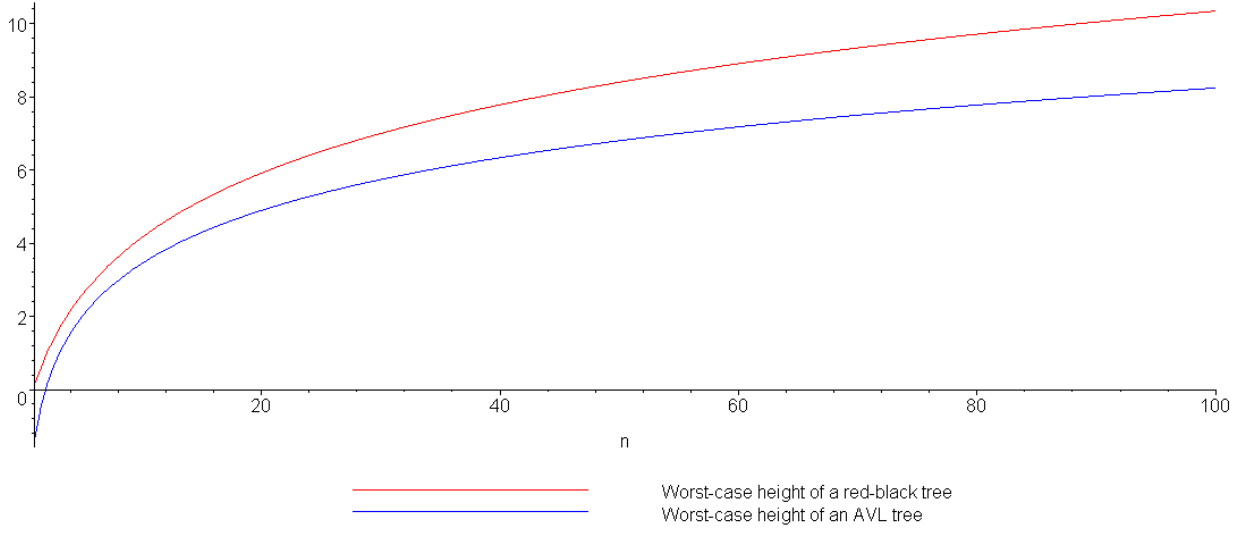
$$h_{\text{worst}} = 2\,\lg(n+2) - 3$$

This grows quicker than the worst-case height for an AVL tree

$$h_{\text{worst}} = \log_{\phi}(n) - 1.3277$$

Red-Black Trees

Plotting the growth of the height of the worst-case red-black tree (red) versus the worst-case AVL tree (blue) demonstrates this:



Red-Black Trees

This table shows the number of nodes in a worst-case trees for the given heights

Thus, an AVL tree with 131070 nodes has a height of 23 while a red-black tree could have a height as large as 31

Height	AVL Tree	Red-Black Tree
1	2	2
3	7	6
5	20	14
7	54	30
9	143	62
11	376	126
13	986	254
15	2583	510
17	6764	1022
19	17710	2046
21	46367	4094
23	121492	8190
25	317810	16382
27	832039	32766
29	2178308	65534
31	5702886	131070
33	14930351	262142

Red-Black Trees

Comparing red-black trees with AVL trees, we note that:

- Red-black trees require one extra bit per node
- AVL trees require one byte per node (assuming the height will never exceed 255)
  - aside: we can reduce this to two bits, storing one of -1, 0, or 1 indicating that the node is left heavy, balanced, or right heavy, respectively

Red-Black Trees

AVL trees are not as deep in the worst case as are red-black trees

- therefore AVL trees will perform better when numerous searches are being performed,
- however, insertions and deletions will require:
  - more rotations with AVL trees, and
  - require recursions from and back to the root
- thus AVL trees will perform worse in situations where there are numerous insertions and deletions

AVL search

RB insert delete

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 not a full node (*i.e.*, two children) must have the same number of black nodes

The first and second rules are local: they affect a node and its neighbours

The third rule 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 node so as to not break the global rule:

- the new node must be red

We will then travel up the tree to the root fixing the requirement that the children of a red node must be black

Bottom-Up Insertions

If the parent of the inserted node is already black, we are done

– Otherwise, we must correct the problem

(parent of inserted is red)

We will look at two cases:

– the initial insertion, and

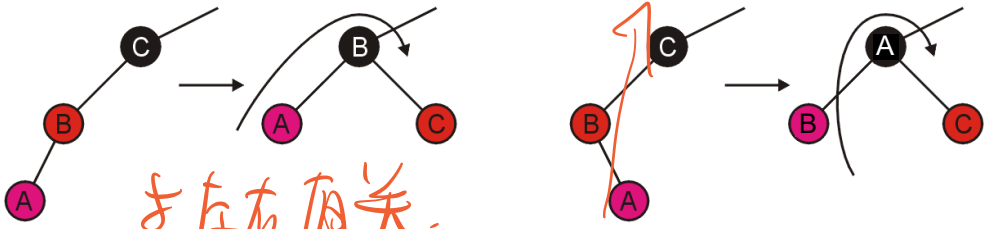
– the recursive steps back to the root

Bottom-Up Insertions

For the initial insertion, there are two possible cases:

- the grandparent has one child (the parent), or
- the grandparent has two children (both red)

Inserting A, the first case can be fixed with a rotation:

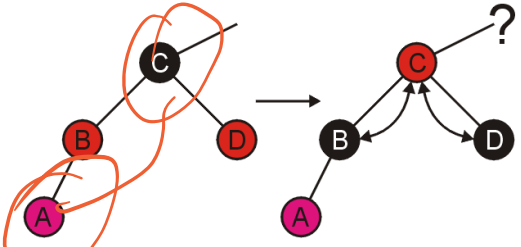


Consequently, we are finished...

SWAP Color

Bottom-Up Insertions

The second case can be fixed more easily, just swap the colours:



Unfortunately, we now may cause a problem between the grandparent and the great grandparent....

G = RC

Bottom-Up Insertions

Fortunately, dealing with problems caused within the tree are identical to the problems at the leaf nodes

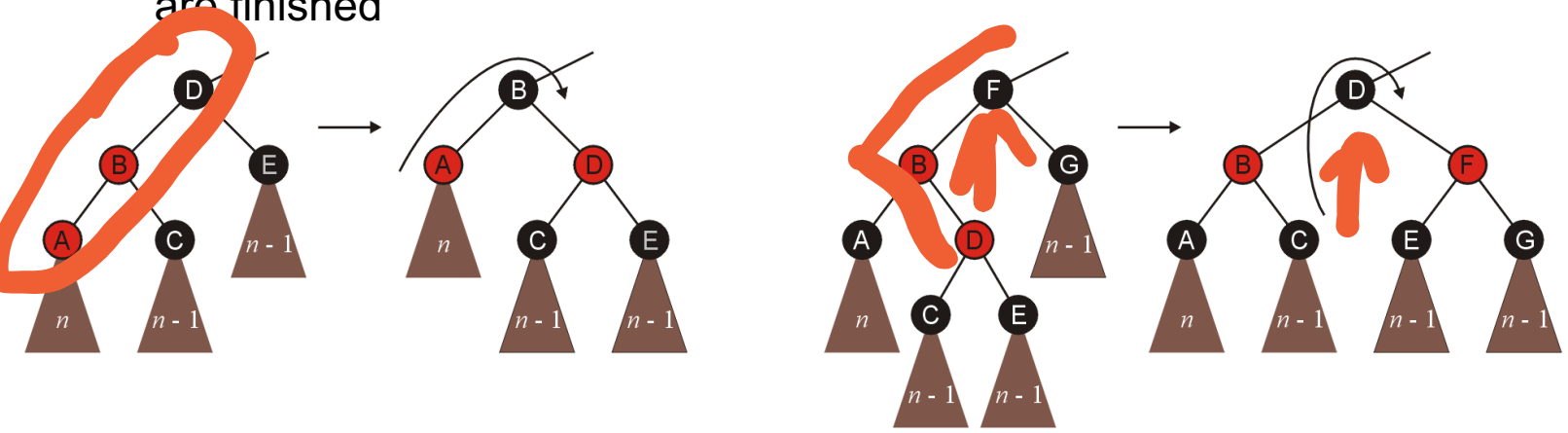
Like before, there are two cases:

- the grandparent has one child (the parent), or
- the grandparent has two children (both red)

Bottom-Up Insertions

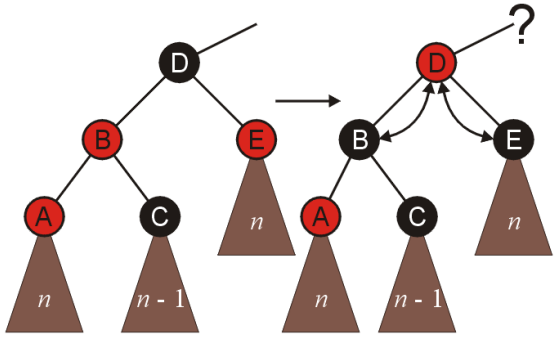
Suppose that A and D, respectively were swapped

In both these cases, we perform similar rotations as before, and we are finished



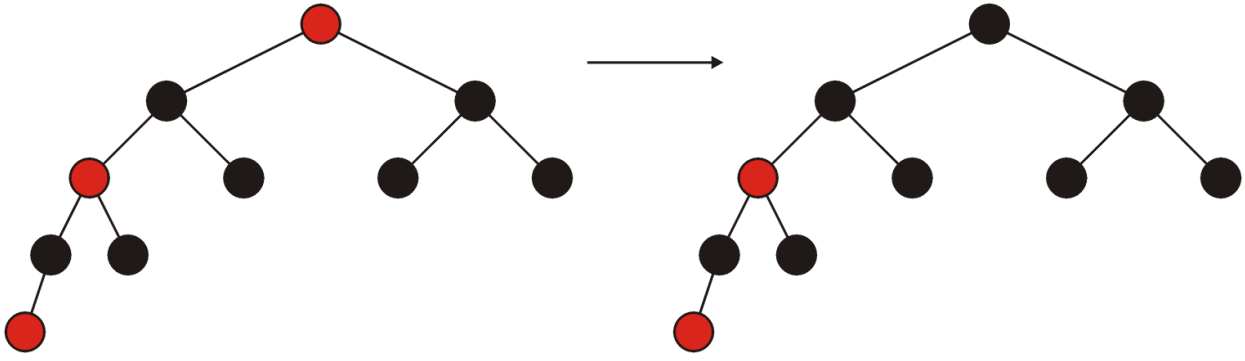
Bottom-Up Insertions

In the other case, where both children of the grandparent are red, we simply swap colours, and recurs back to the root



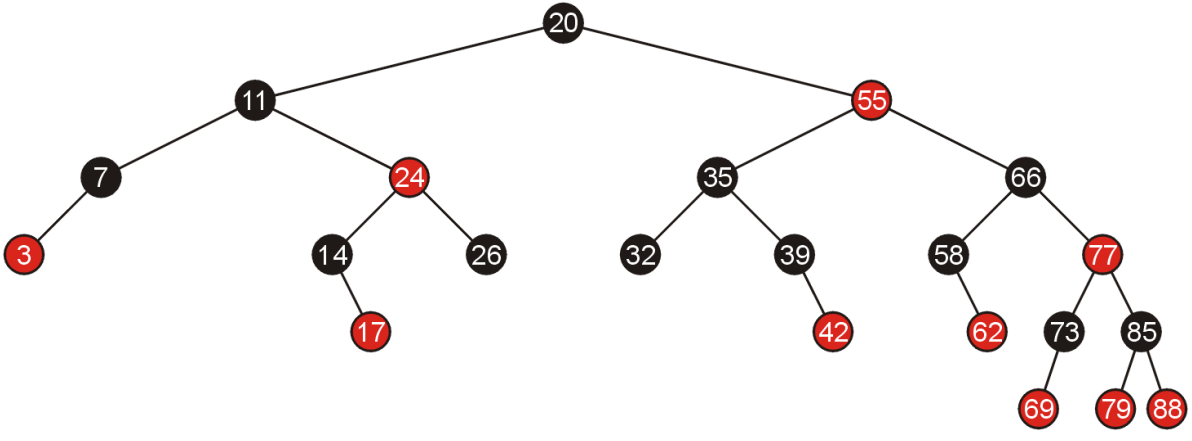
Bottom-Up Insertions

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



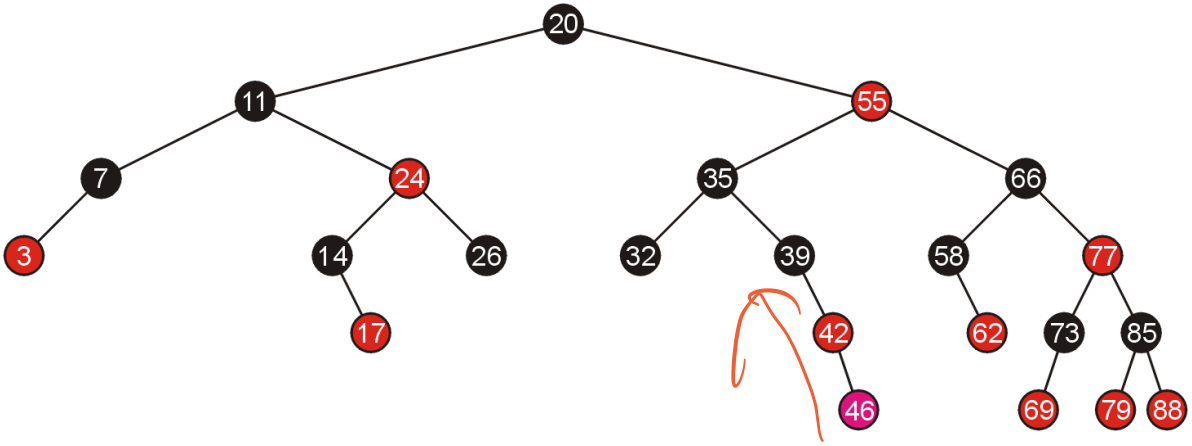
Examples of Insertions

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



Examples of Insertions

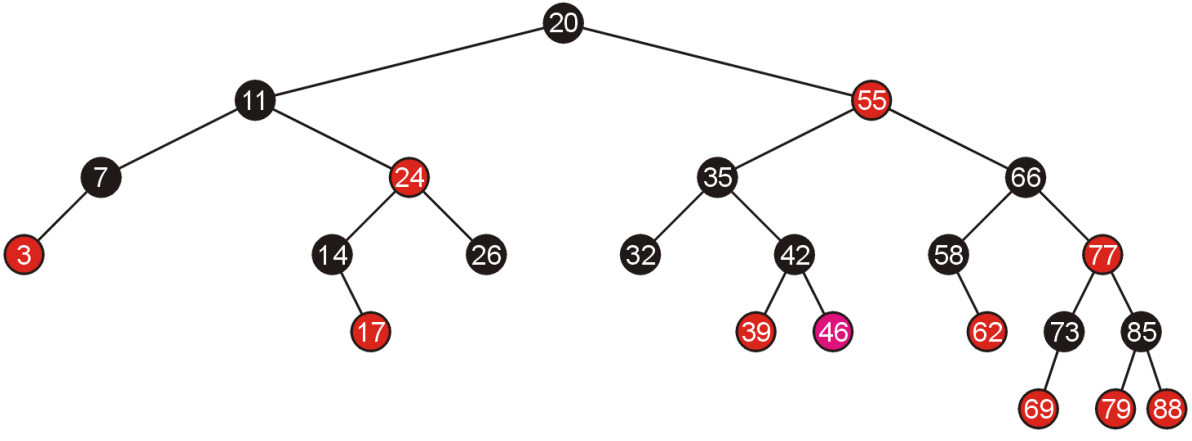
Adding 46 creates a red-red pair which can be corrected with a single rotation



插入结点保持 AVL 特点

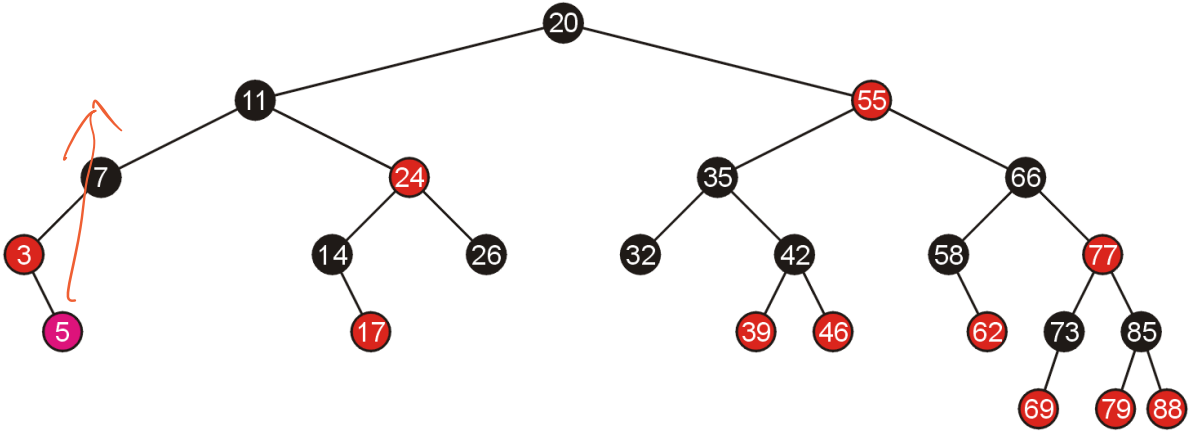
Examples of Insertions

Because the pivot is still black, we are finished



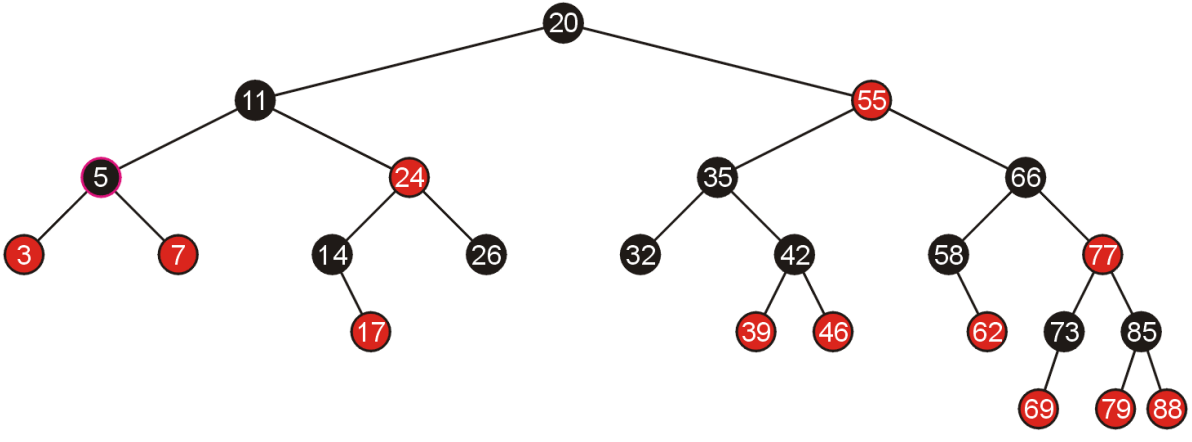
Examples of Insertions

Similarly, adding 5 requires a single rotation



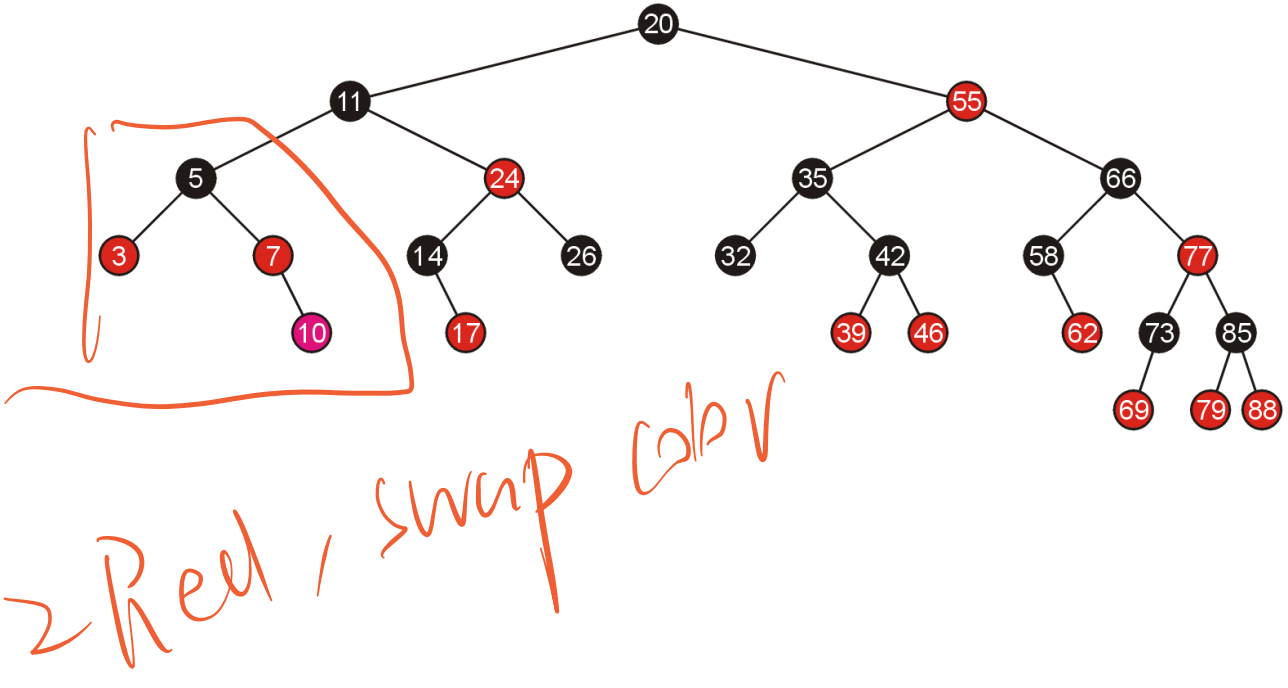
Examples of Insertions

Which again, does not require any additional work



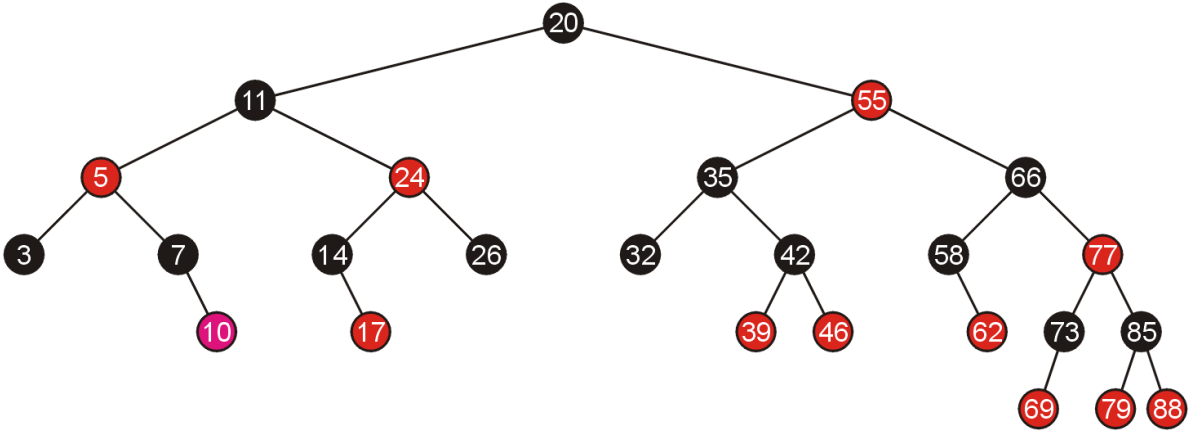
Examples of Insertions

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



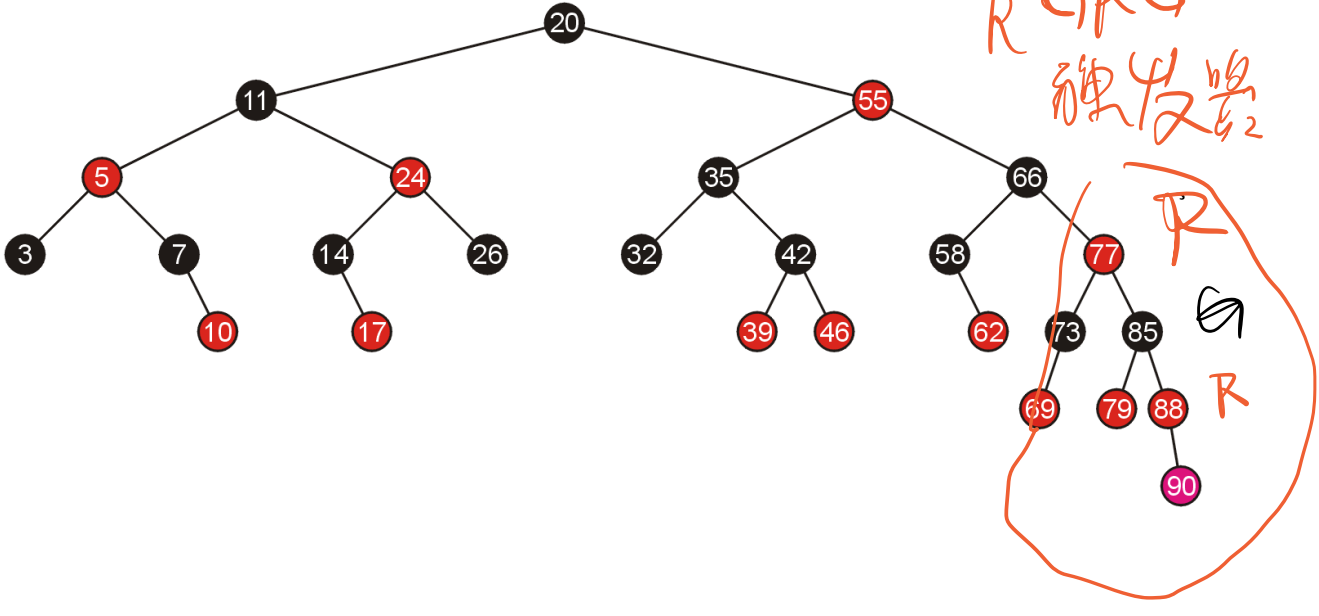
Examples of Insertions

Because the parent of 5 is black, we are finished



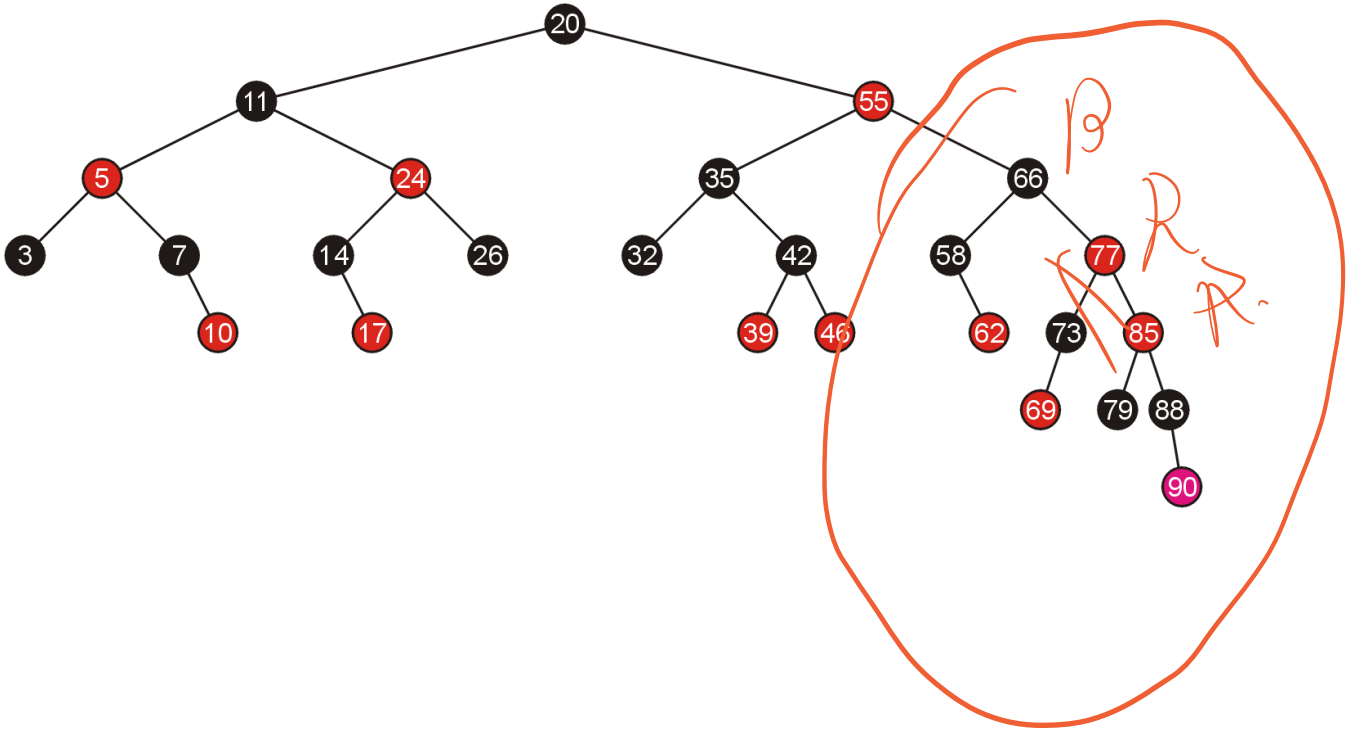
Examples of Insertions

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



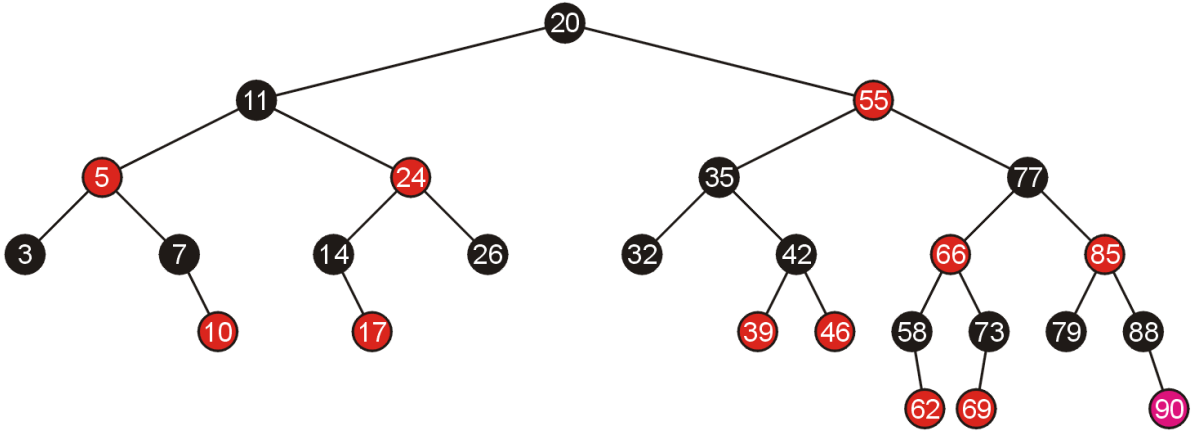
Examples of Insertions

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



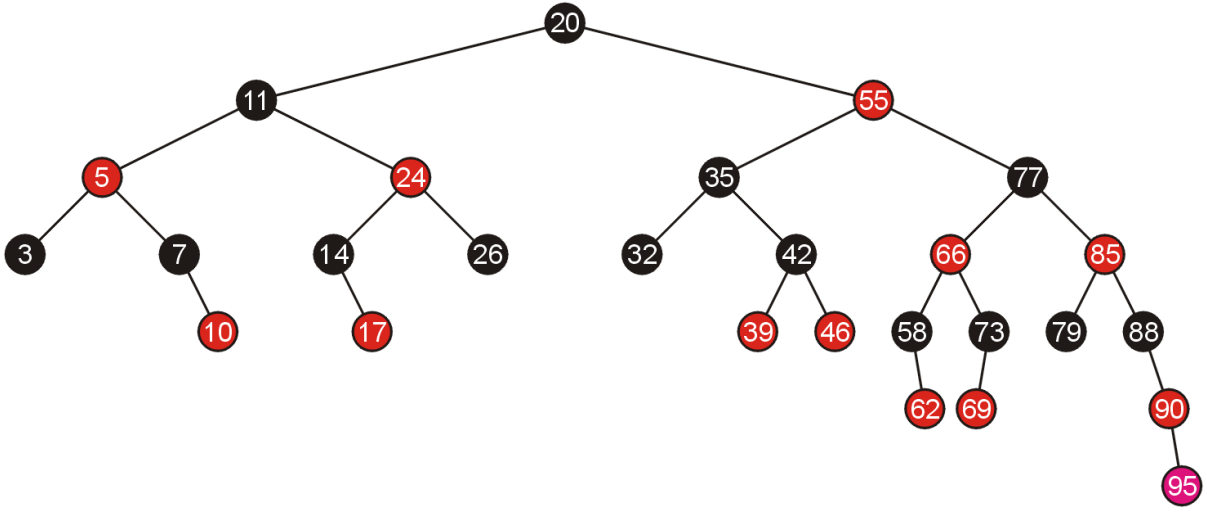
Examples of Insertions

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



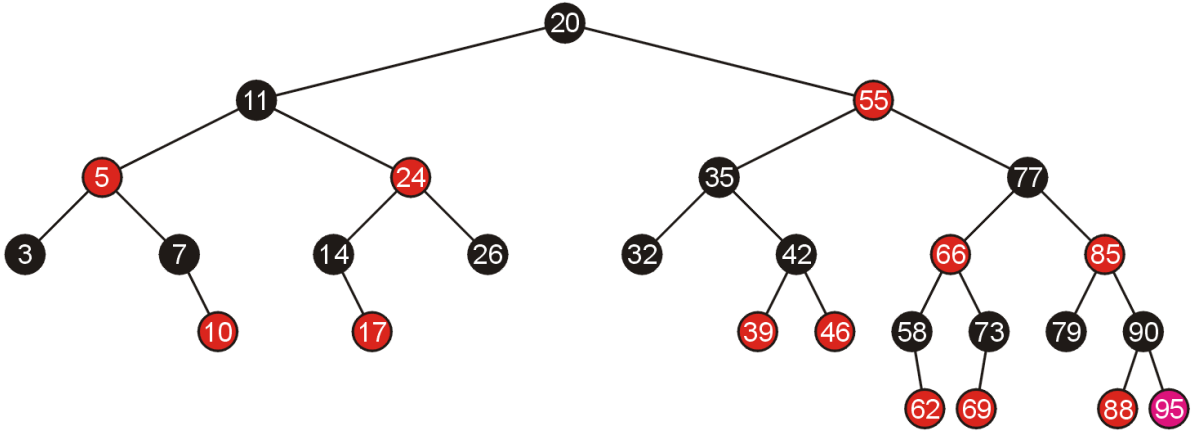
Examples of Insertions

Inserting 95 requires a single rotation



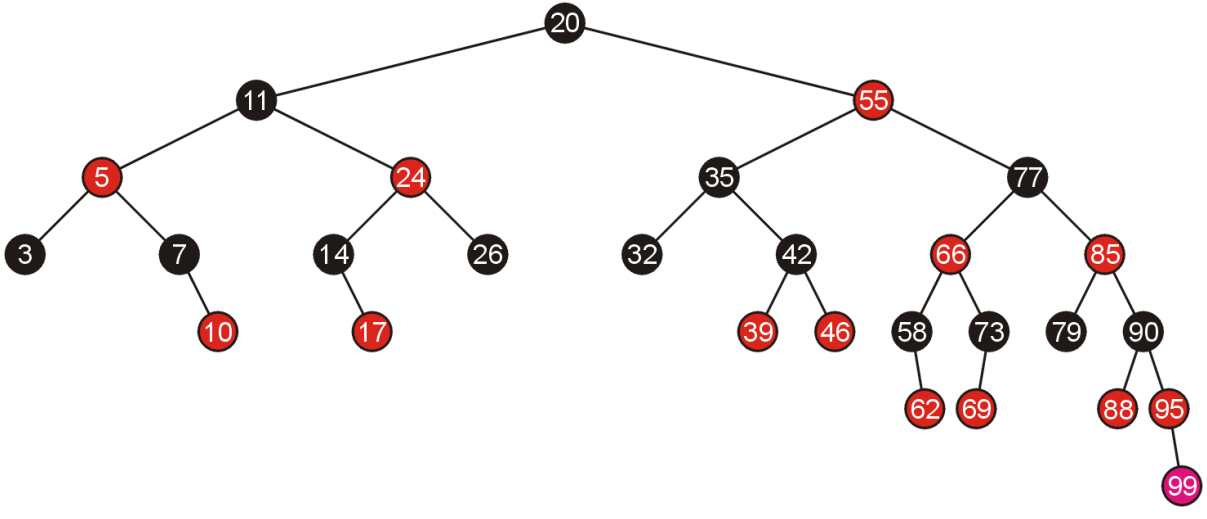
Examples of Insertions

And consequently, we are finished



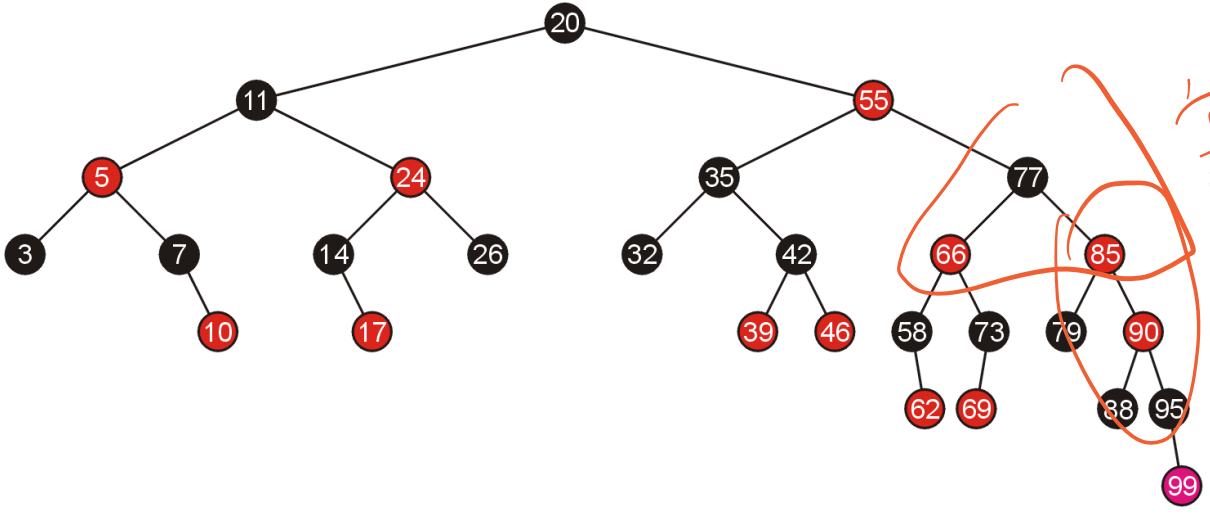
Examples of Insertions

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



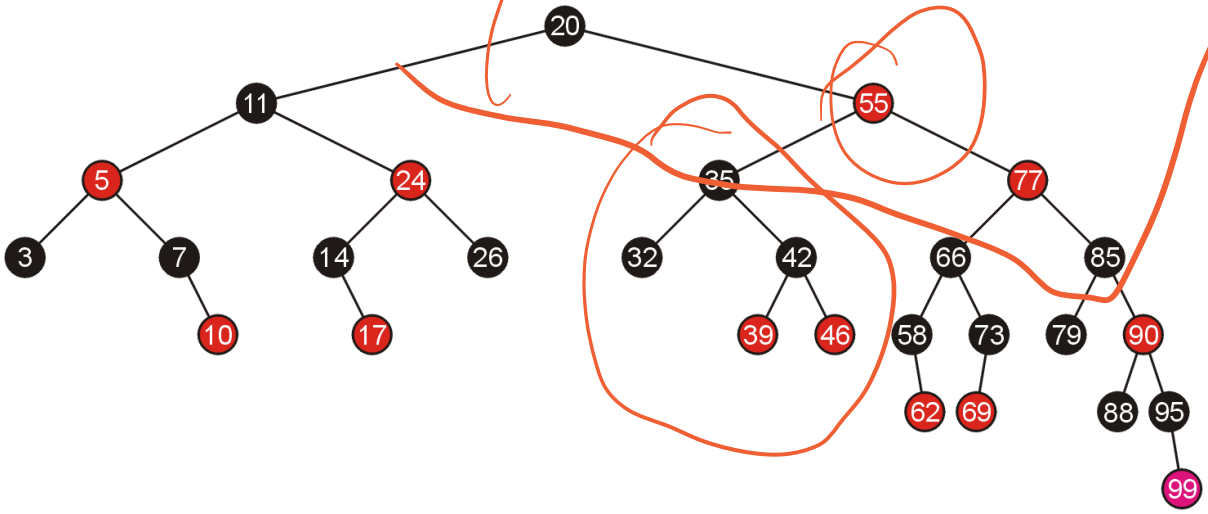
Examples of Insertions

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



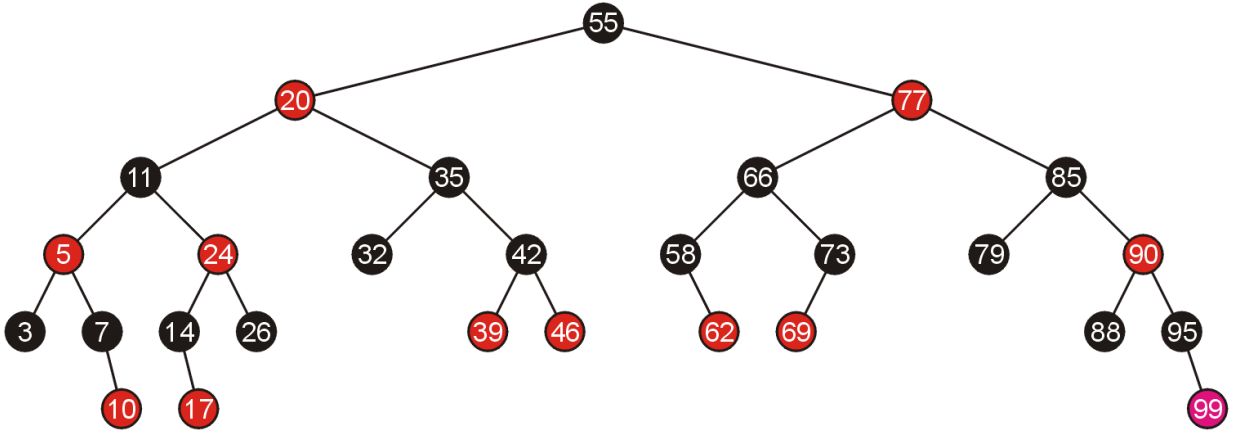
Examples of Insertions

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



Examples of Insertions

Thus, the rotation solves the problem



Top-Down Insertions and Deletions

With a bottom-up insertion, it is first necessary to search the tree for the appropriate location, and only then recurs back to the root correcting any problems

- This is similar to AVL trees

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

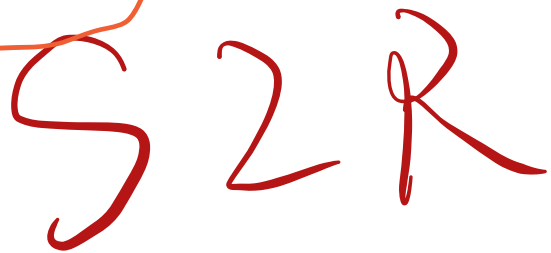
Top-Down Insertions

The important observation is:

- swapping may require recursive corrections going back all the way to the root
- rotations do not require recursive steps back to the root

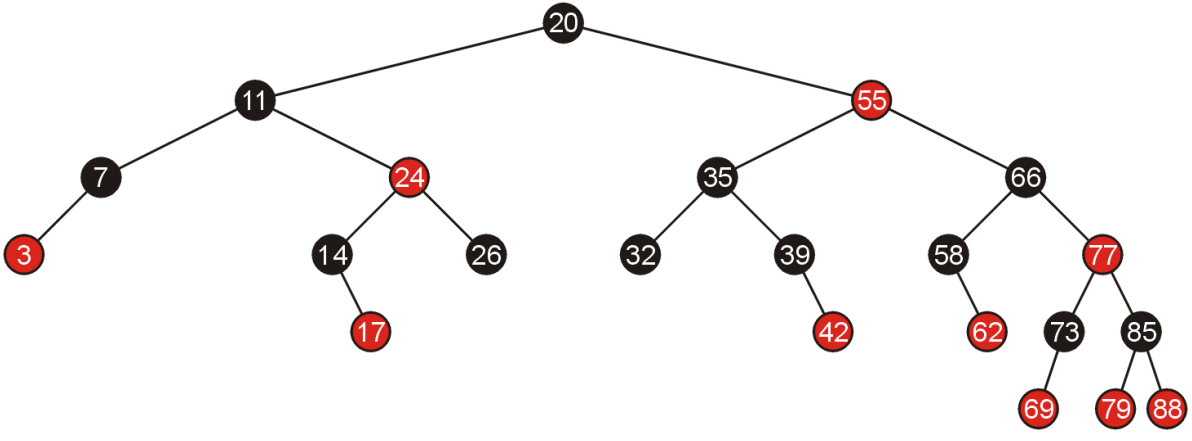
Therefore, while moving down from the root, automatically swap the colours of any black node with two red children

- this may require at most one rotation at the parent of the now-red node



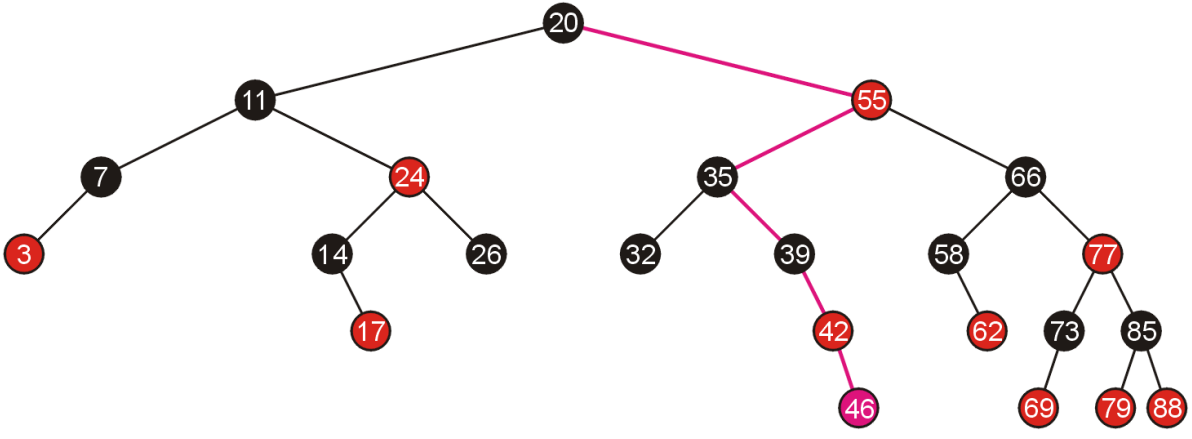
Examples of Top-Down Insertions

We will start with the same red-black tree as before, but make top-down insertions (no recursion):



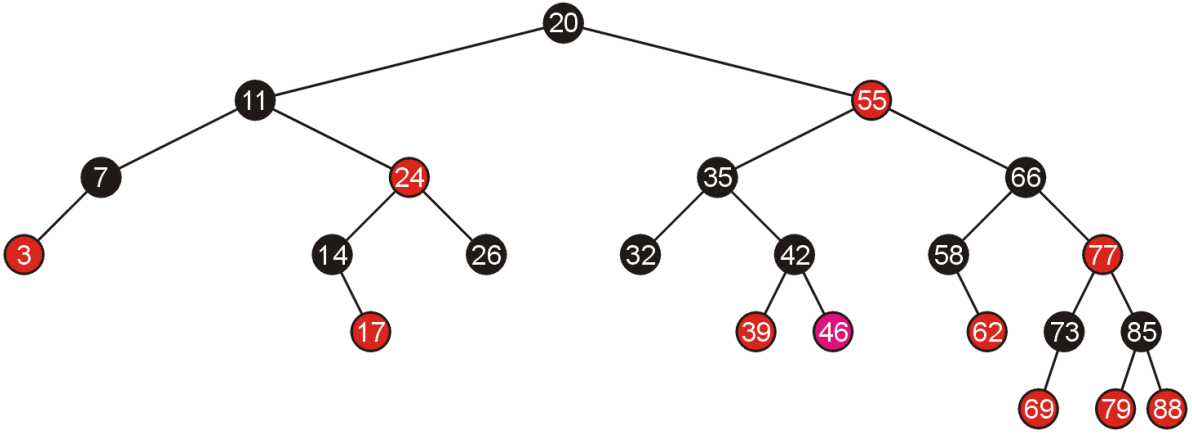
Examples of Top-Down Insertions

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



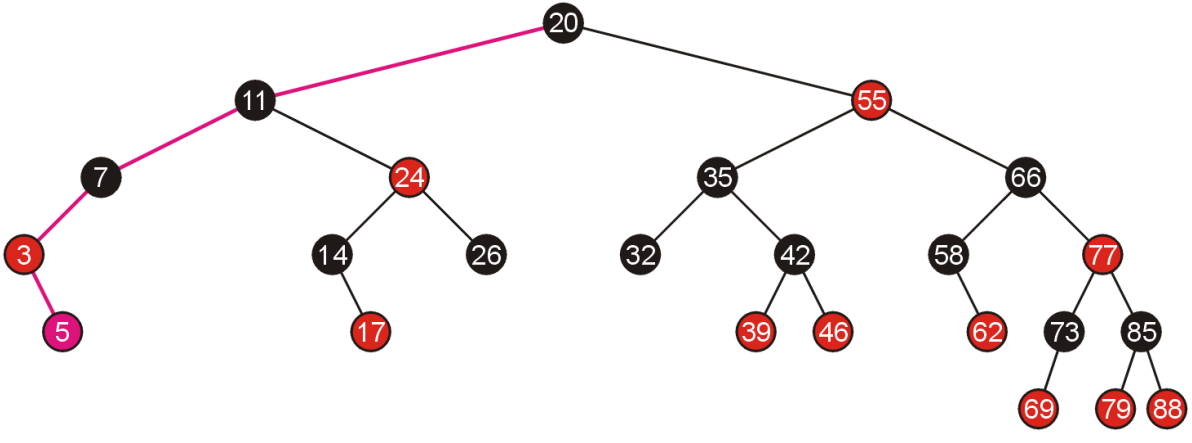
Examples of Top-Down Insertions

However, it does require one rotation at the end



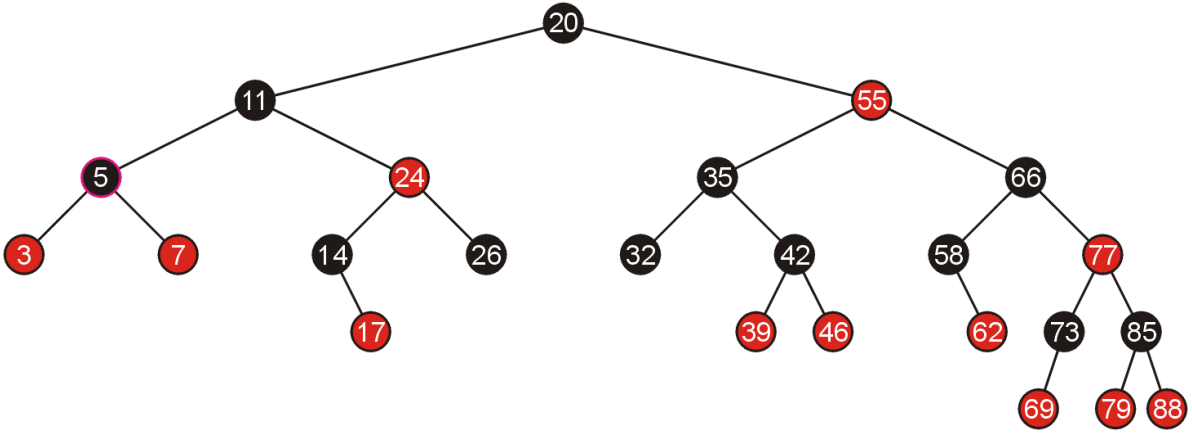
Examples of Top-Down Insertions

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



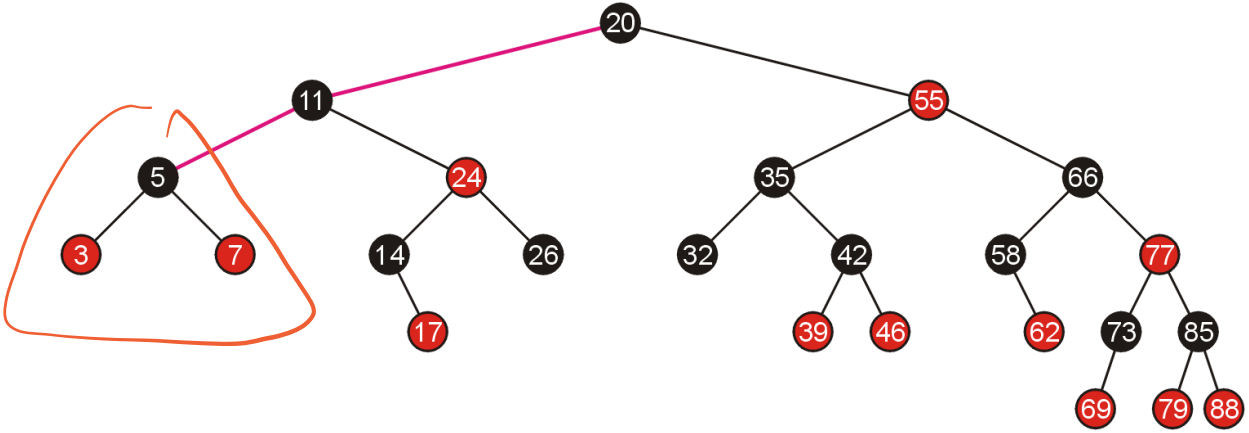
Examples of Top-Down Insertions

A rotation solves the last problem



Examples of Top-Down Insertions

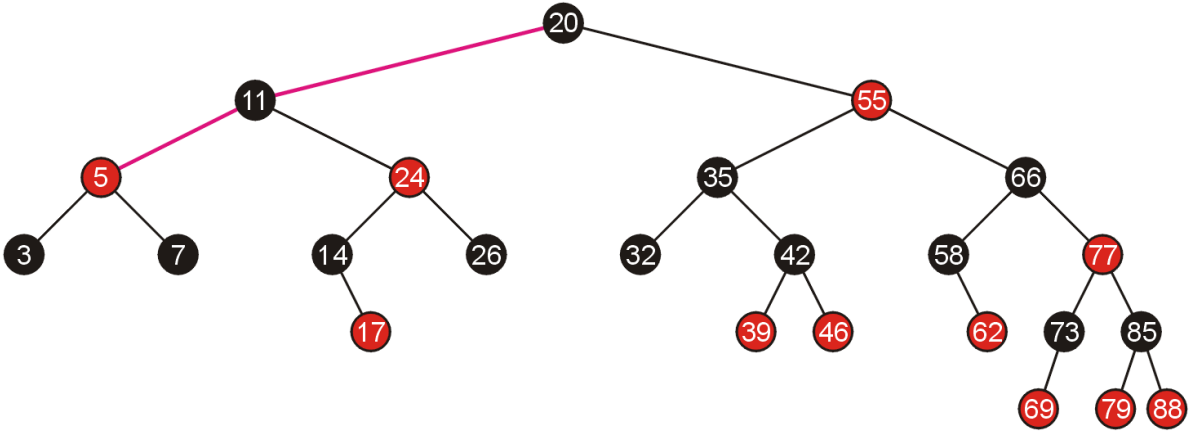
Adding 10 causes us to search down two edges before we meet node 5 with two red children



Swap

Examples of Top-Down Insertions

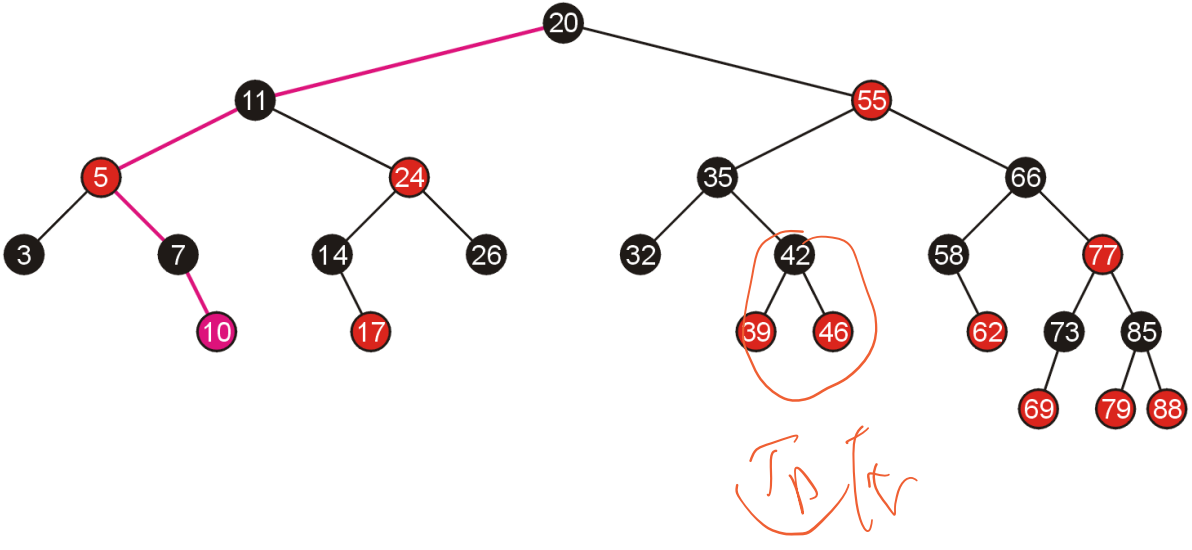
We swap the colours, and this does not cause a problem between 5 and 11



Examples of Top-Down Insertions

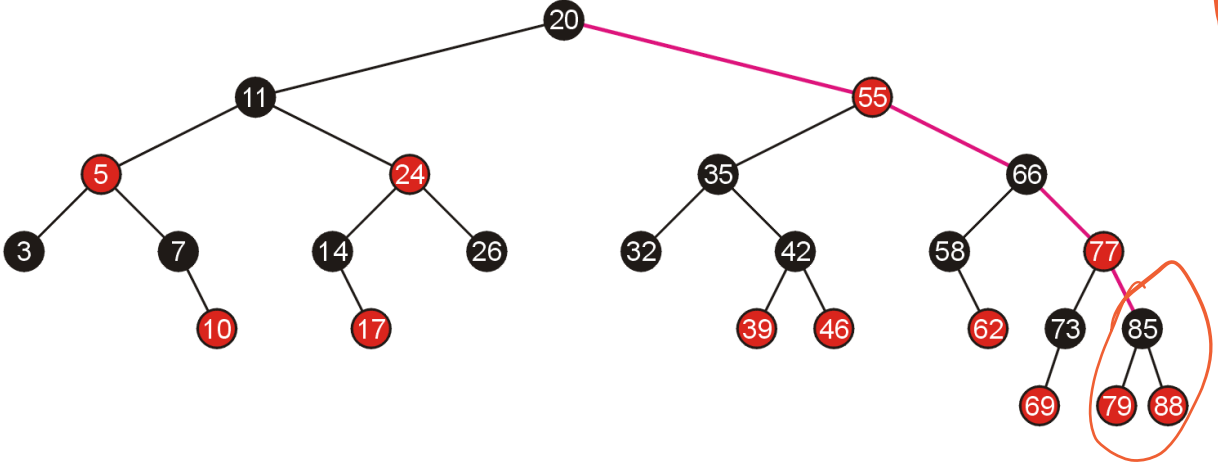
We continue and place 10 in the appropriate location

– No further rotations are required



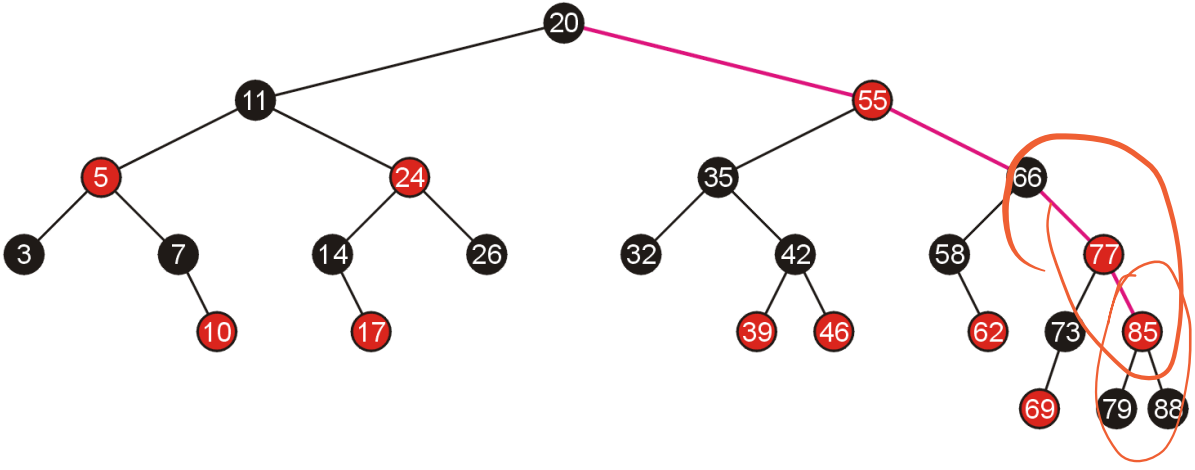
Examples of Top-Down Insertions

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



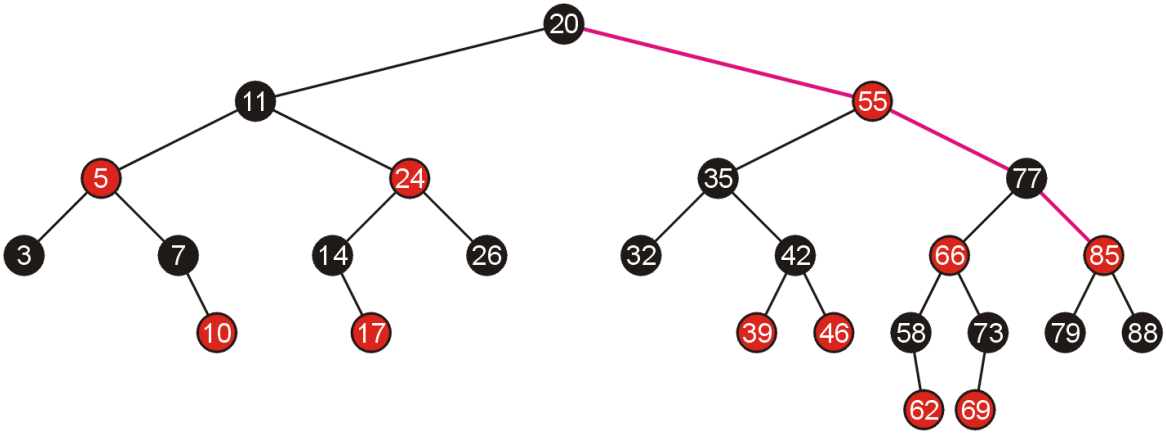
Examples of Top-Down Insertions

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



Examples of Top-Down Insertions

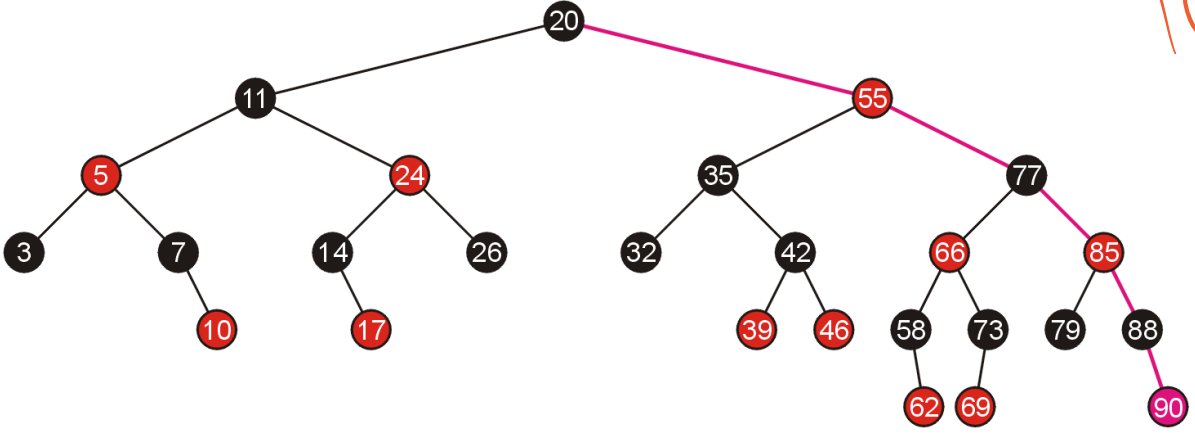
We require only one rotation to solve this problem, and we are guaranteed that this will not cause any problem for its parents



Examples of Top-Down Insertions

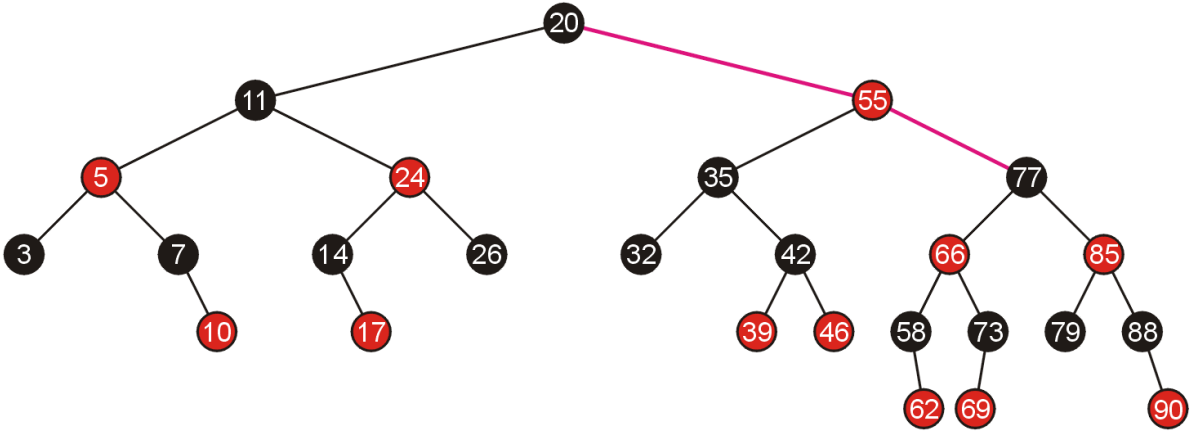
We continue to search down the right path and add 90 in the appropriate location—no further corrections are required

last



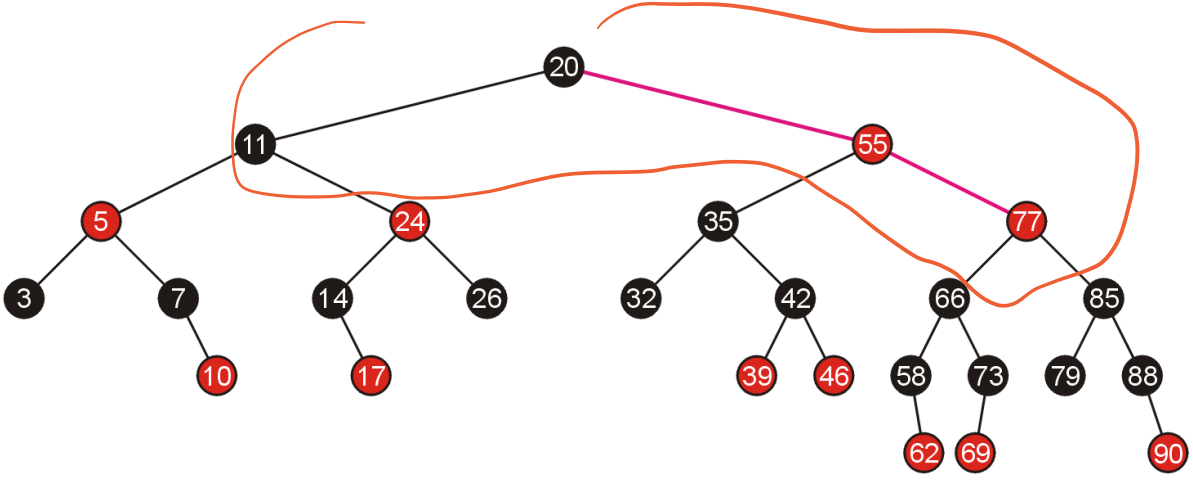
Examples of Top-Down Insertions

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



Examples of Top-Down Insertions

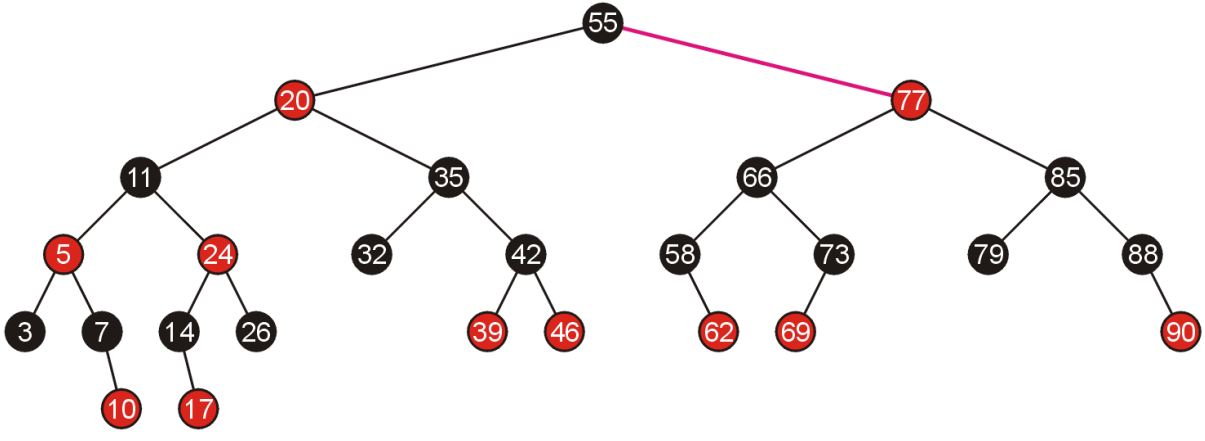
We swap the colours, which causes a red-red parent-child combination which must be fixed by a rotation



Examples of Top-Down Insertions

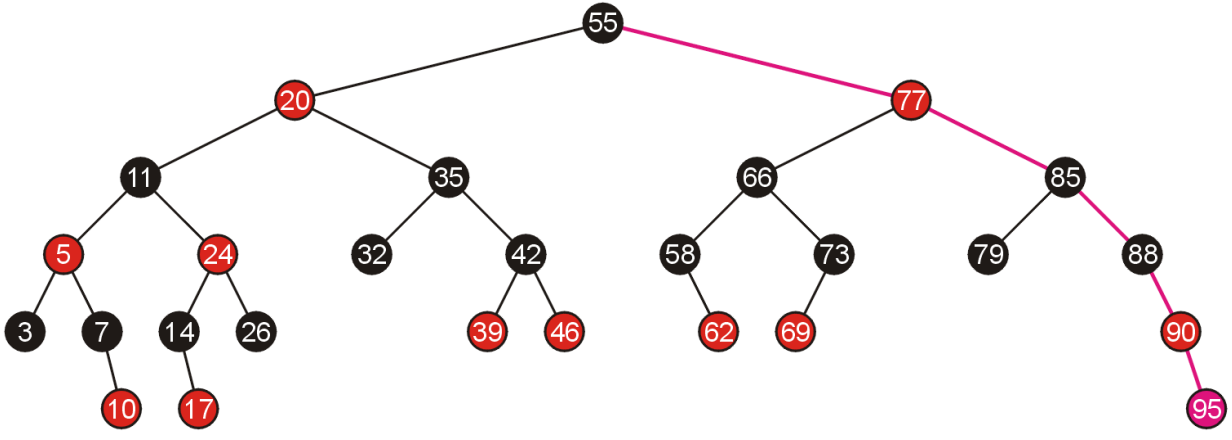
The rotation is around the root

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



Examples of Top-Down Insertions

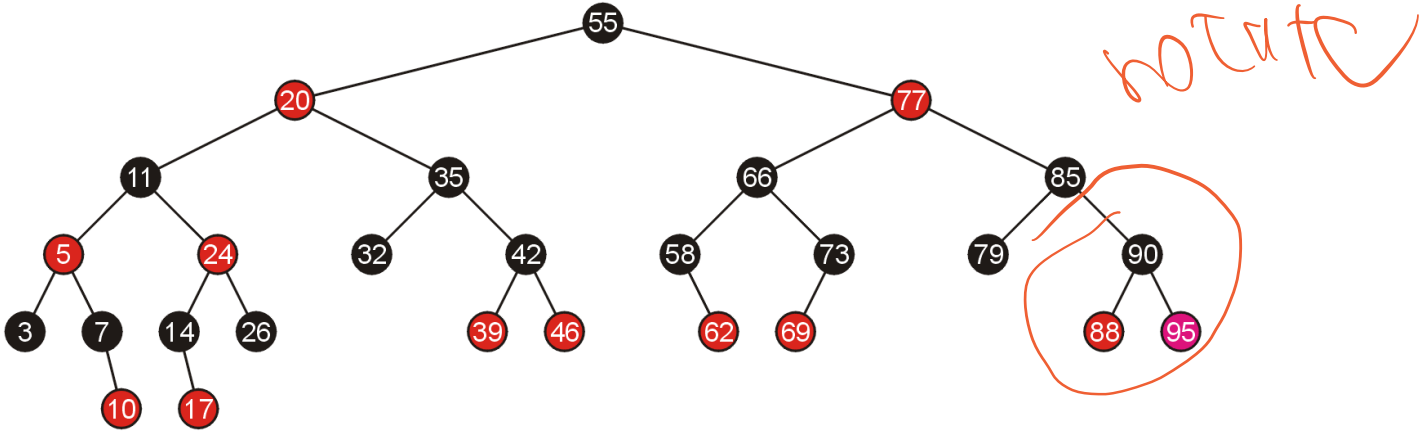
We can now proceed to add 95 by following the right-hand branch, and the insertion causes a red-red parent-child combination



Examples of Top-Down Insertions

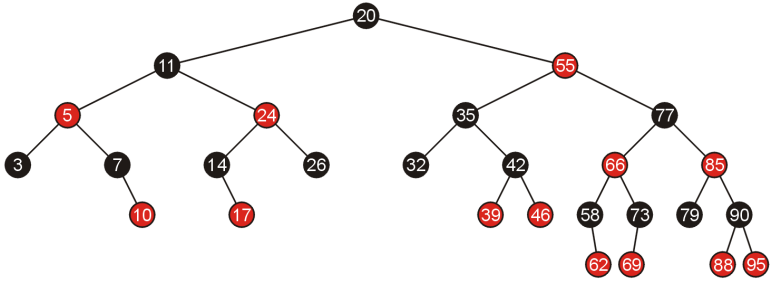
This is fixed with a single rotation

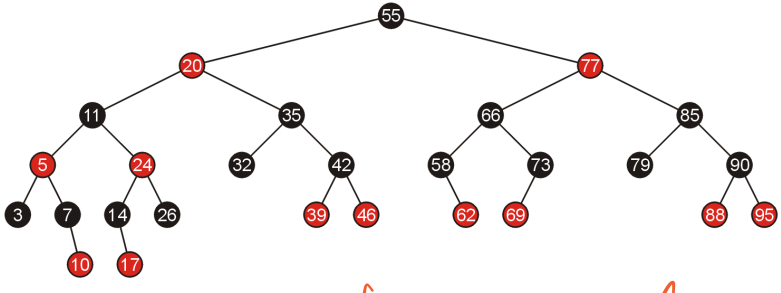
– We are guaranteed that this will not cause any further problems



Examples of Top-Down 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



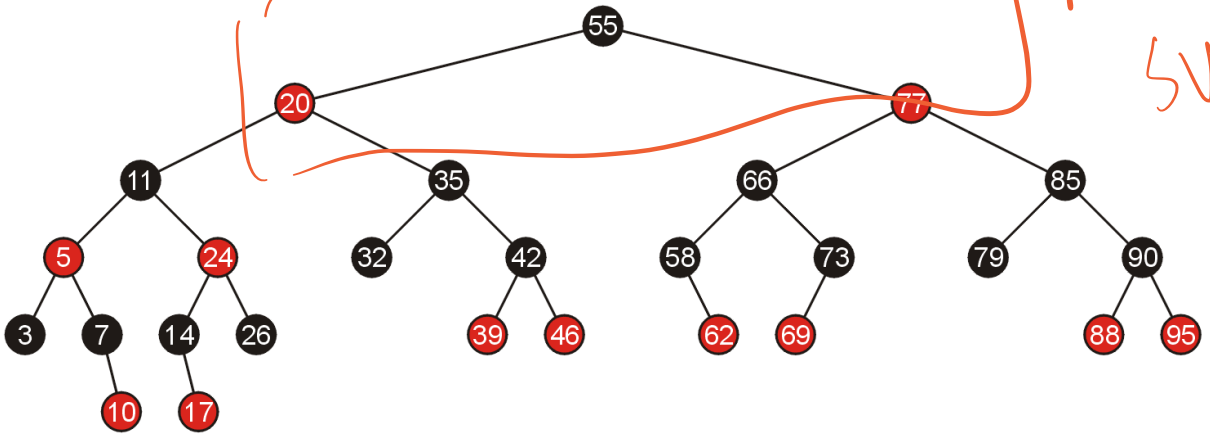


bottom up

top down

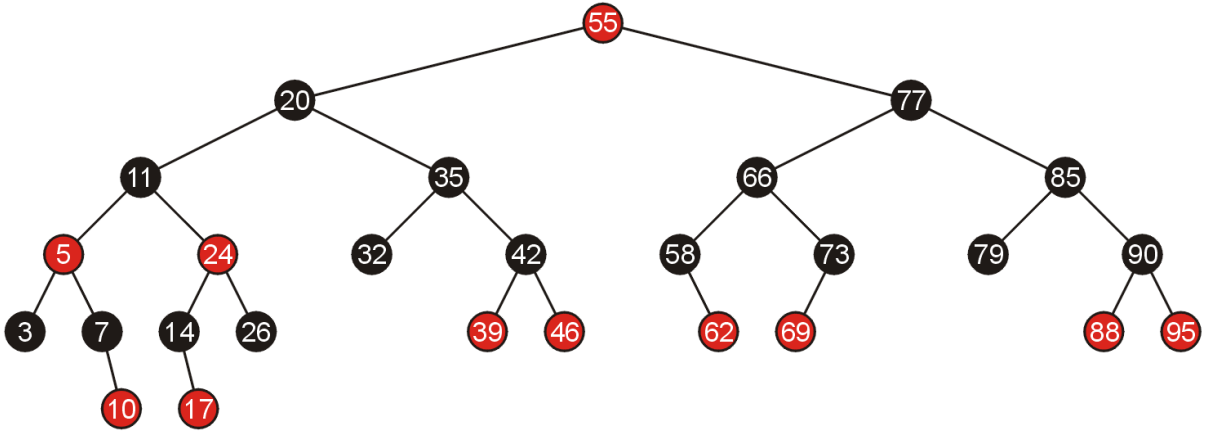
Examples of Top-Down Insertions

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

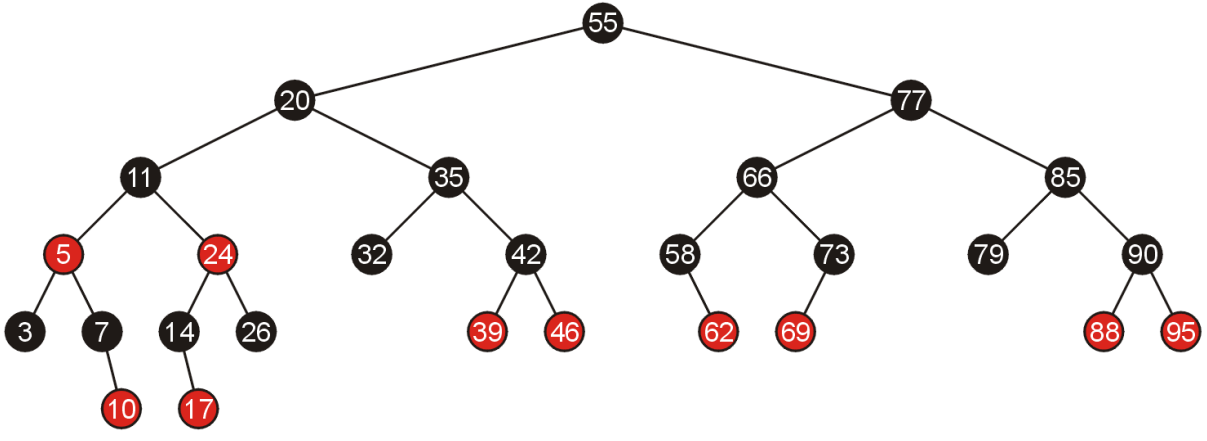
At this point, each path to a non-full node still has the same number of black nodes, however, we violate the requirement that the root is black



Examples of Top-Down Insertions

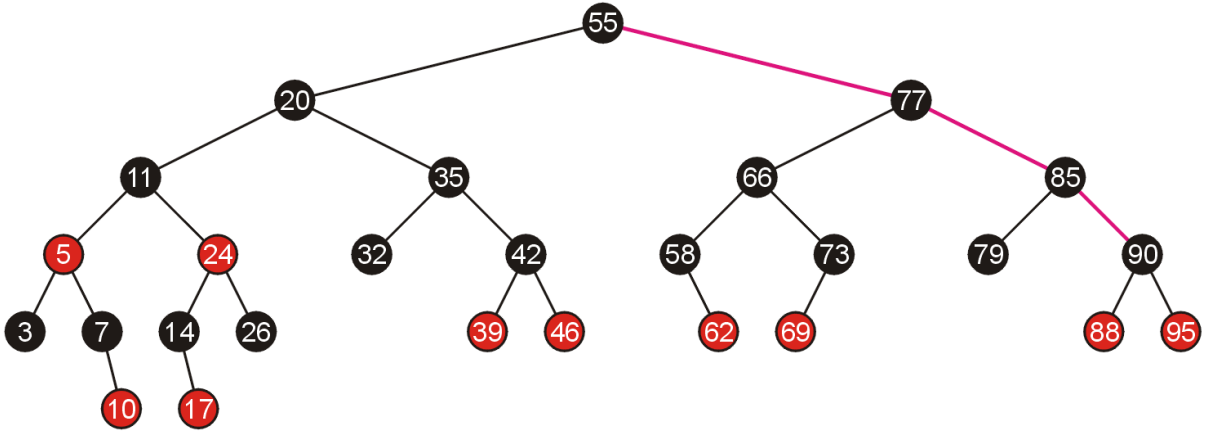
We change the colour of the root to black

– This adds one more black node to each path



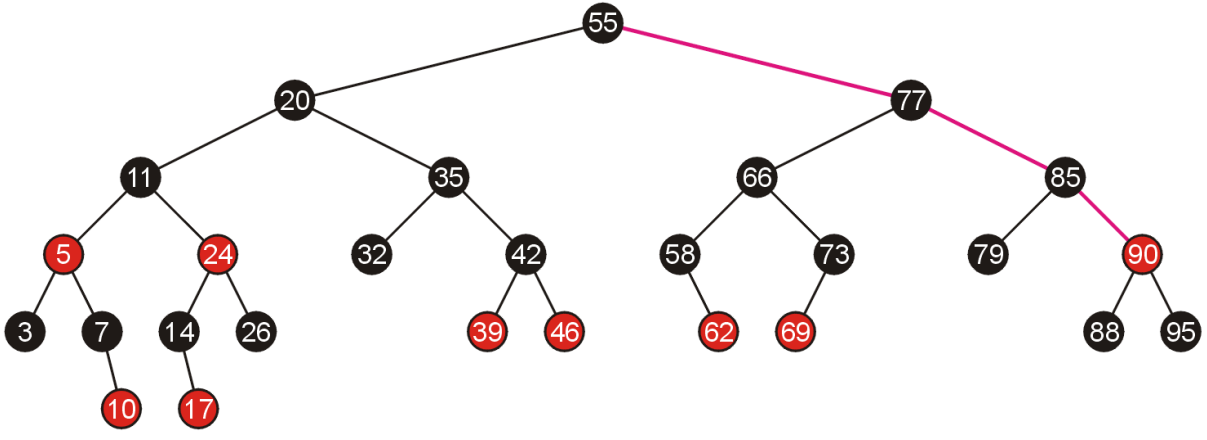
Examples of Top-Down Insertions

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



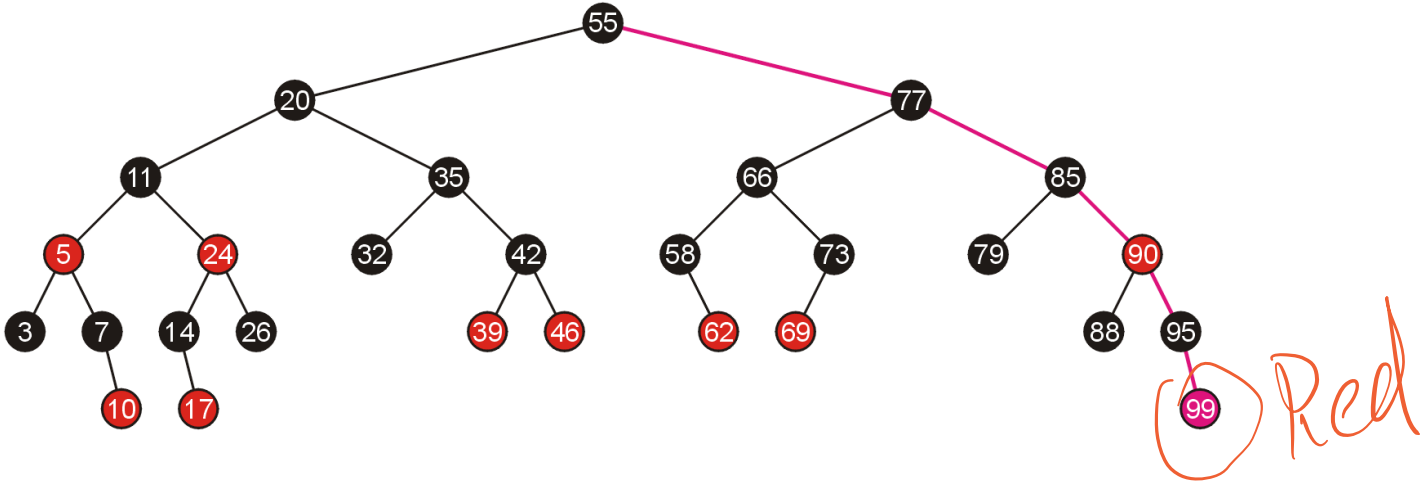
Examples of Top-Down Insertions

We continue down the right to add 99



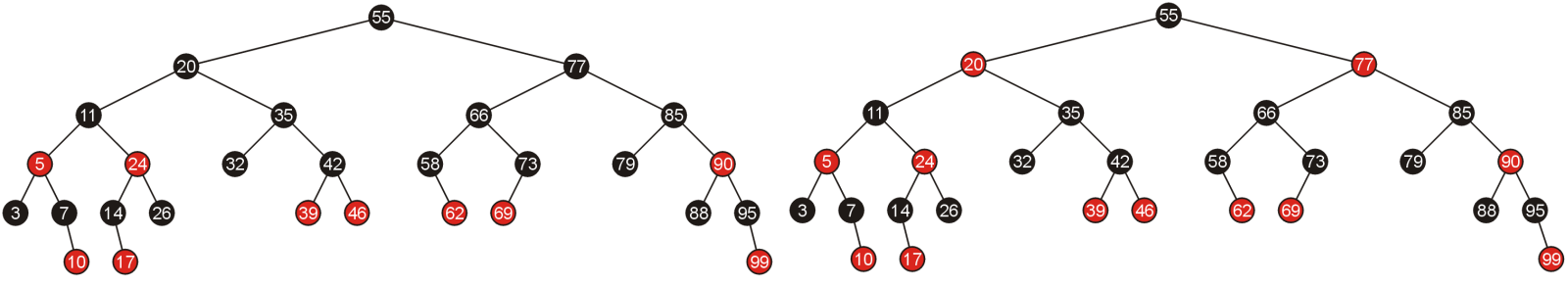
Examples of Top-Down Insertions

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



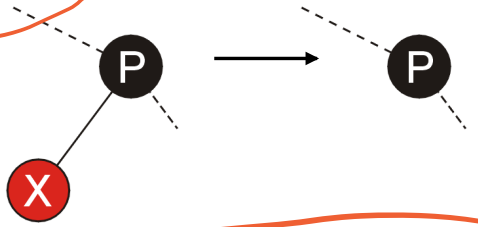
Examples of Top-Down Insertions

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

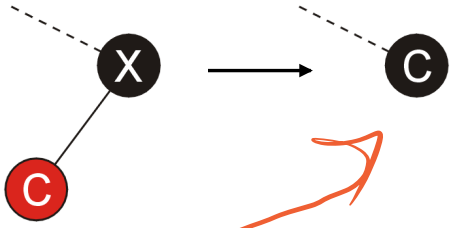


Top-Down Deletions

If we are deleting a red leaf node X, then we are finished



If we are deleting a node X with one child, we only need to replace the value of the deleted node with the value of the leaf node



black

Top-Down Deletions

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

- replace the node with the minimum element in the right sub-tree
- then delete that element from the right sub-tree

Top-Down Deletions

That minimum element must be either:

- a red leaf node,
- a black node with a single red leaf node, or
- a black leaf node

The first two cases are solved, consequently, we need only deal with the possibility that the leaf node we are deleting is black

black leaf delete

Top-Down Deletions

Similar to top-down insertions, we will adopt a strategy which ensures that the leaf node being deleted is not black (check textbook Ch13)

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

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 2009, Ch.13, p.308-338.

[2] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §12.2, p.525-34.

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.