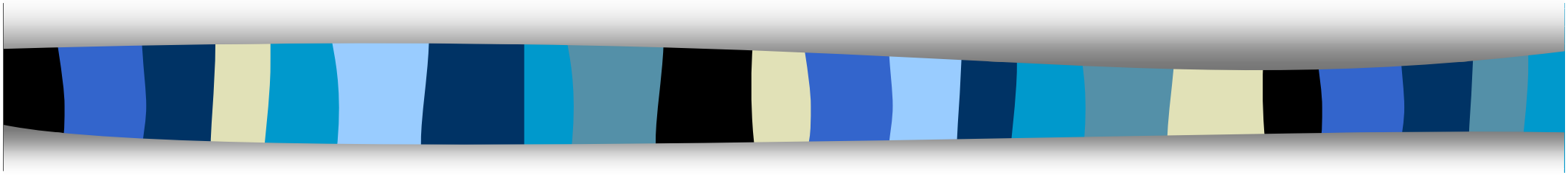
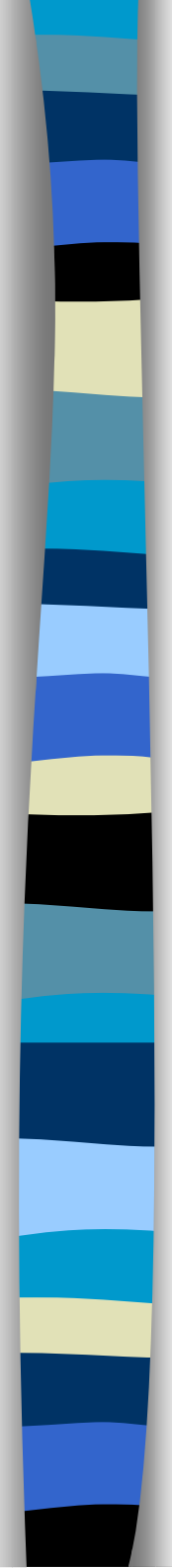
Red Black Balanced Trees



Curtin University of Technology
Department of Computing
Data Structures and Algorithms 120



Topics

- Review of binary tree complexity analysis
- Benefits of keeping trees balanced
- Random Insertion
- Red-Black Trees

Objectives

- Learn the rules used by Red-Black trees to ensure reasonable balanced tree.
- Learn how to apply the Red-Black switch, flip and rotate transforms to change the tree.
- Understand when to apply a given transform to balance the tree.

Reading

Read all of LaFore (1998) Chapter 8, 9.

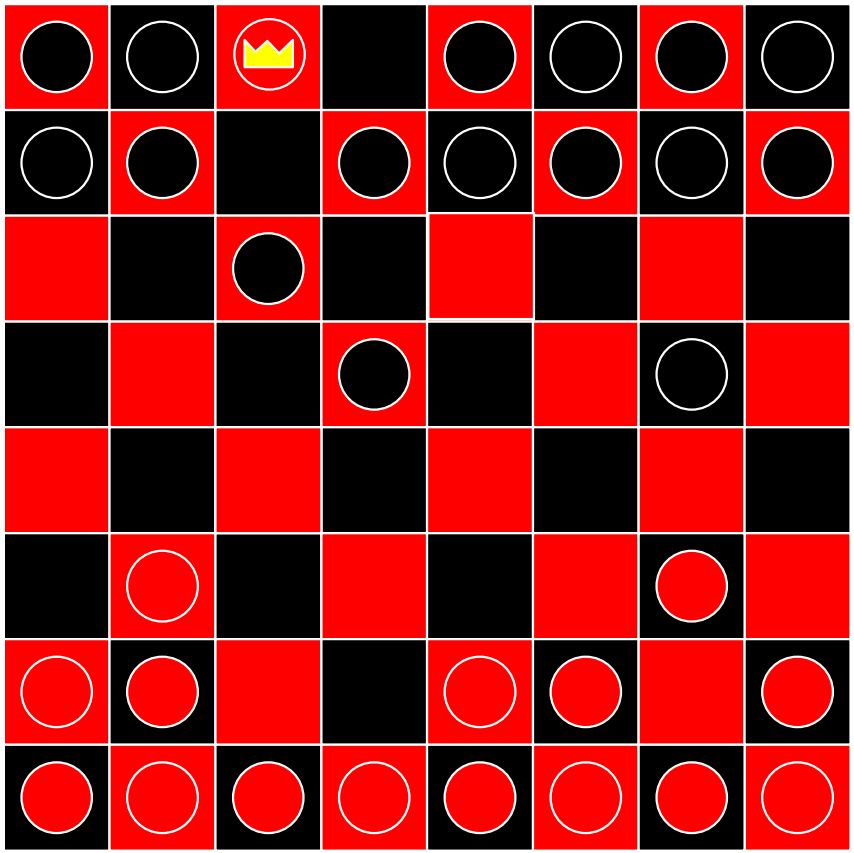
Types of binary trees

- Binary trees classifications:
  - strictly binary tree
  - complete binary tree
  - almost complete binary tree
- Classification is important when evaluating efficiency of processing

Maintaining balance

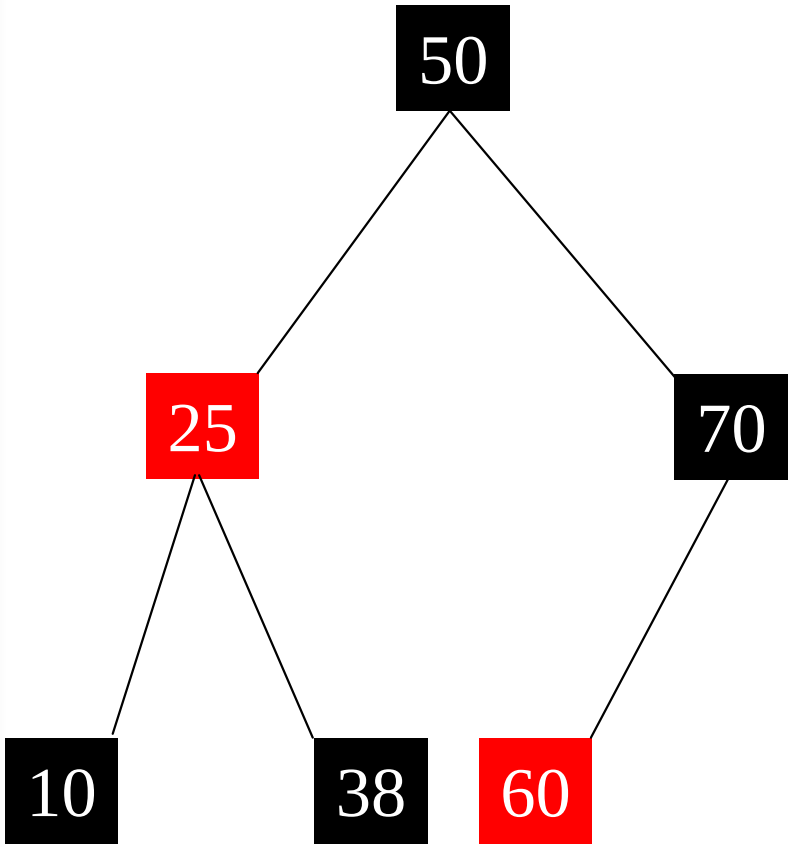
- Maintaining a perfectly balanced tree can be complex and is difficult to maintain.
- Usually follow a set of rules to get us “reasonably” close to being completely balanced.
- These rules will not necessarily produce a perfectly balanced tree.
- The “Red-Black Trees” is one such approach.

Red-Black Checkerboards...



Much as the pattern of red an black squares determine legal move on the rectangular grid of a checkerboard, nodes in a Red-Black Tree are given either a red and a black colour to enforce valid tree geometry.

Red-Black Trees...



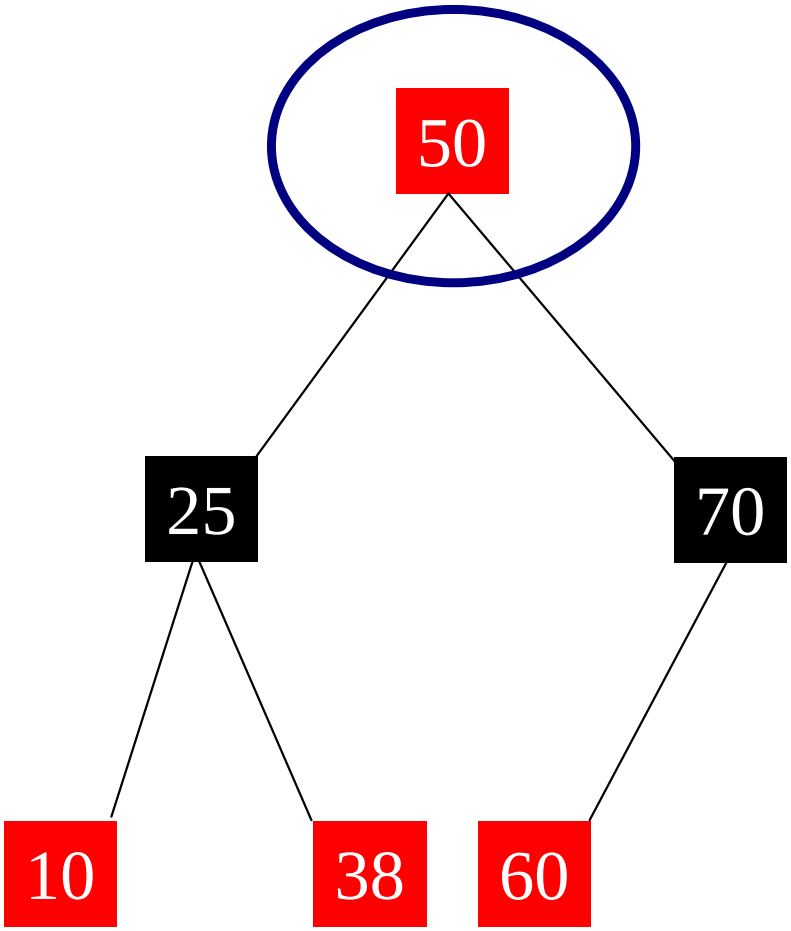
(1) COLOUR RULE:
Every node is either red or black.

(2) BLACK ROOT RULE:
The root is always black

(3) RED PARENT RULE:
If a node is red, it's children MUST be black. However, if a node is black, it's children aren't required to be be red.

(4) BLACK HEIGHT RULE:
Every path from *root* to *leaf* or a *null child* must contain the same number of black nodes.

Violation?



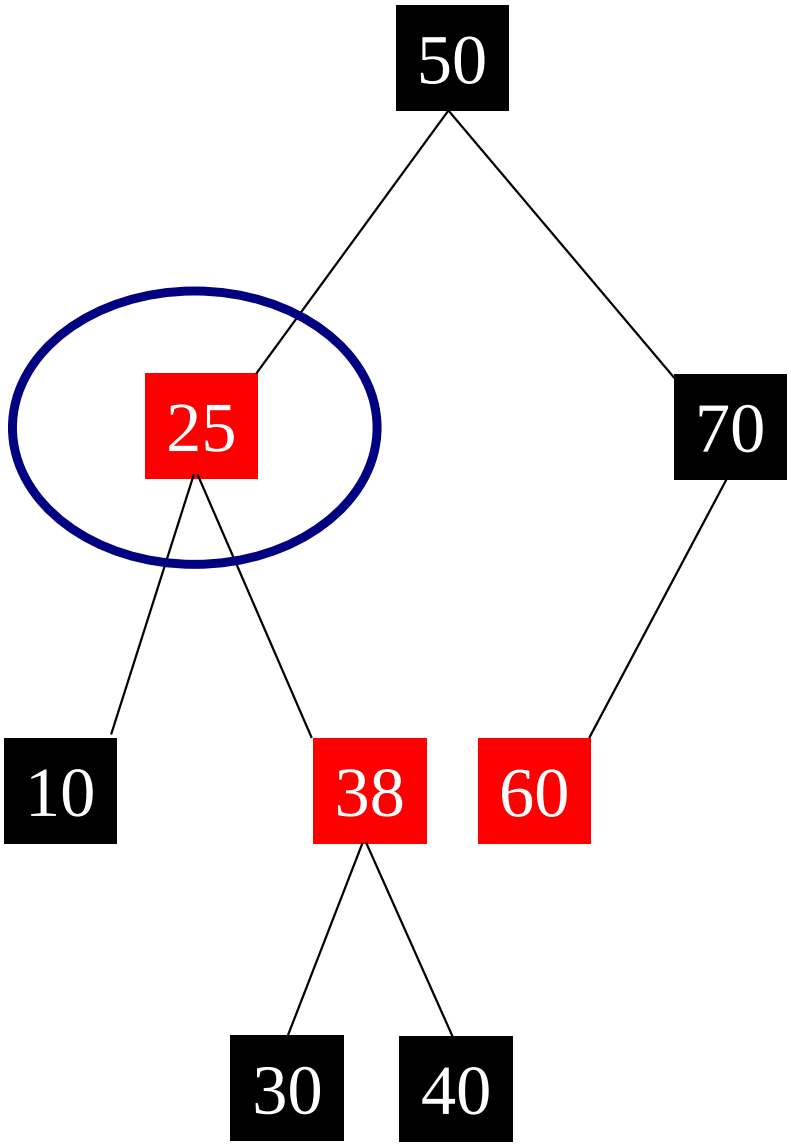
☑COLOUR RULE:
Every node is either red or black.

☒BLACK ROOT RULE:
The root is always black

☑RED PARENT RULE:
If a node is red, it's children MUST be black. However, if a node is black, it's children aren't required to be be red.

☑BLACK HEIGHT RULE:
Every path from *root* to *leaf* or a *null child* must contain the same number of black nodes.

Violation?



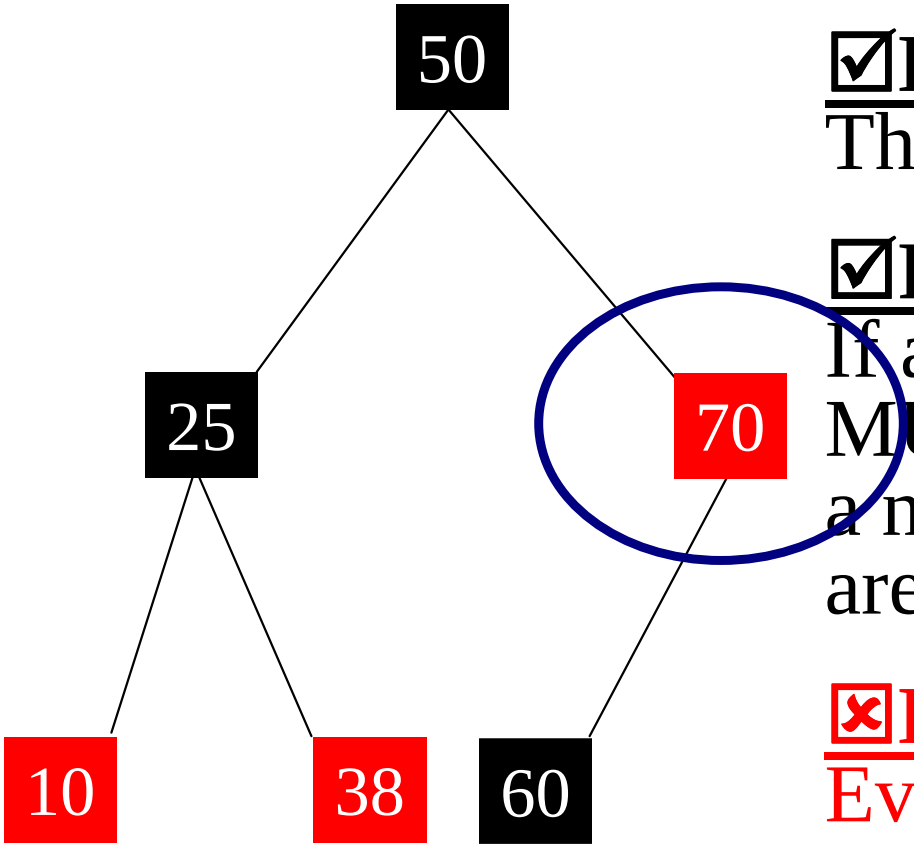
☑COLOUR RULE:
Every node is either red or black.

☑BLACK ROOT RULE:
The root is always black

☒RED PARENT RULE:
If a node is red, it's children MUST be black. However, if a node is black, it's children aren't required to be be red.

☑BLACK HEIGHT RULE:
Every path from *root* to *leaf* or a *null child* must contain the same number of black nodes.

Violation?



☑COLOUR RULE:
Every node is either red or black.

☑BLACK ROOT RULE:
The root is always black

☑RED PARENT RULE:
If a node is red, it's children MUST be black. However, if a node is black, it's children aren't required to be be red.

☒BLACK HEIGHT RULE:
Every path from root to leaf or a ***null child*** must contain the same number of black nodes.

When inserting...

- New nodes are always coloured red.
- This will minimise rule violations since new nodes are always leaf nodes with no children.
- Doing so will not violate rules 1, 2, or 4, but may sometimes violate rule 3 (The Red Parent Rule).

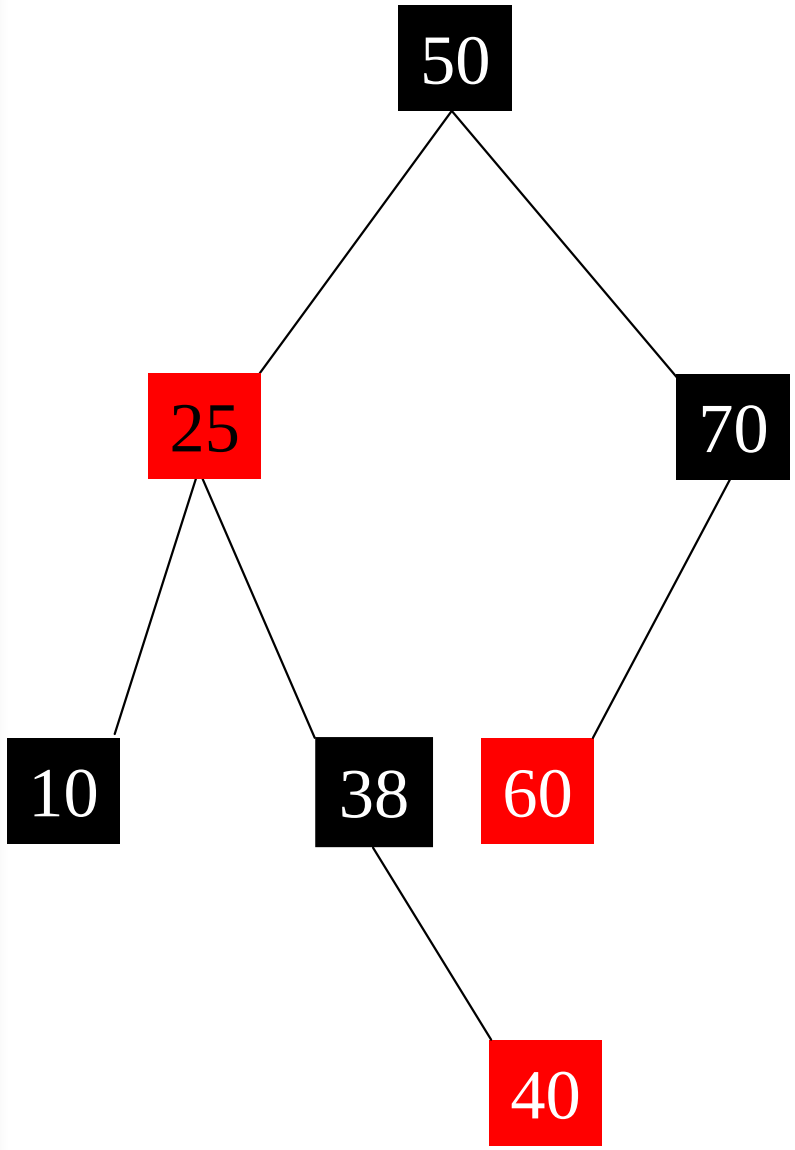
What happens if the rules are violated?

- If an operation results in a tree that is not “red-black correct”, it must be “fixed” before or after insertion.
- What can be done?
  1. Flipping colours of the parent and children can sometimes help fix a potential rule violation.
  2. Can also switch colour of a single node.
  3. Rotating and possibly grafting sub-trees into new positions can help fix a potential rule violation.

Why flip?

- Flips can be used to turn red leaf nodes into black leaf nodes.
- This is useful, since new nodes are always inserted as red leaf nodes.
- Another useful property of flips is that they leave the black height the same!
- Black node with red children are flipped on the way down.

Insert 40



Can’t insert 40, needs colour flip at node 25.

Flipping nodes doesn’t violate rules.

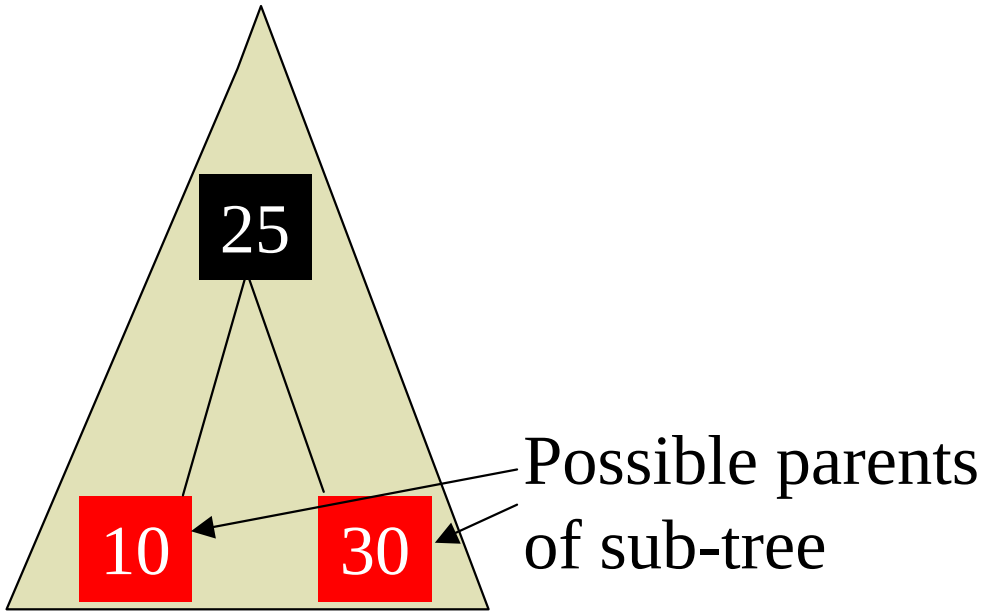
Can now insert 40 without violating any rules

Tree isn’t perfectly balanced, but isn’t “too” bad

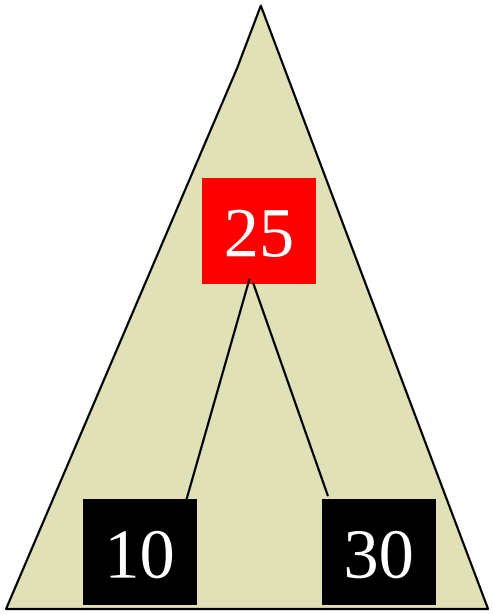
Flips don’t effect black height

Assume five more black nodes above sub-tree, then black height is six both before and after flip!

If the parent of the sub-tree is red, then we flip colours **ON THE WAY DOWN <u>BEFORE</u>** the new node is inserted.

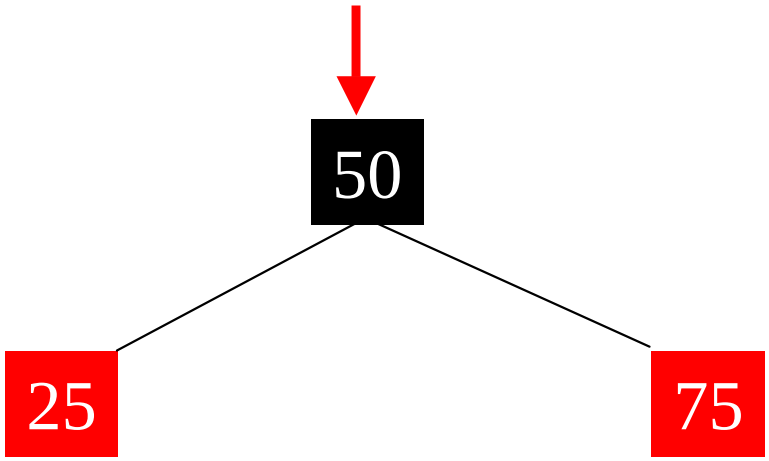


<u>Can’t</u> insert a new red leaf node



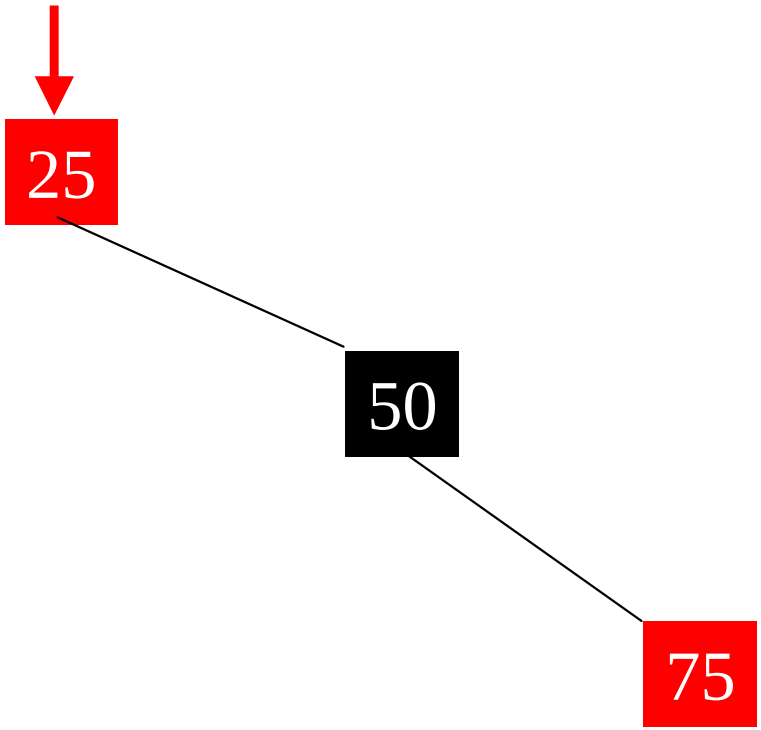
<u>Can</u> insert a new red leaf node

So what's a rotate?



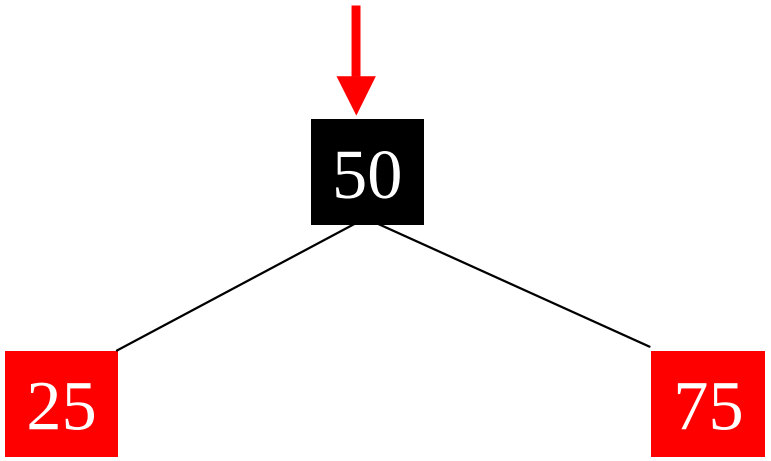
This initial tree violates no rules, but we'll do some rotations anyway to get the idea.

Result of rotate right...



This rotation violates a rule!
Can you tell which one?

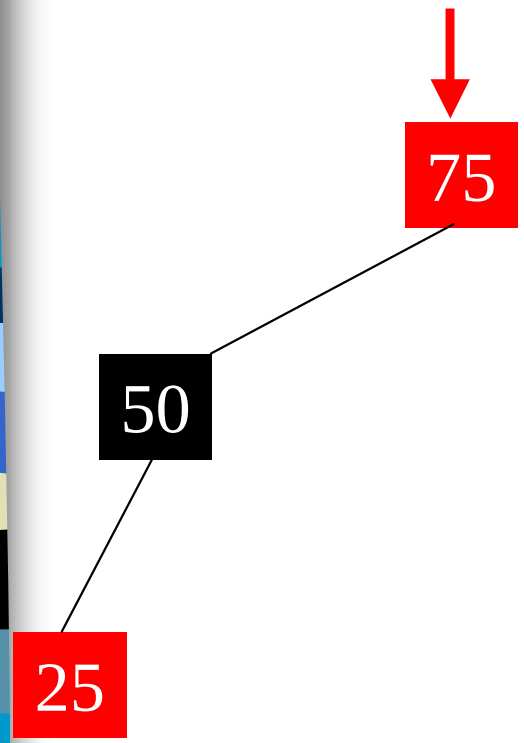
Result of subsequent rotate left



We’re back where we started!

The tree is now red-black correct once again

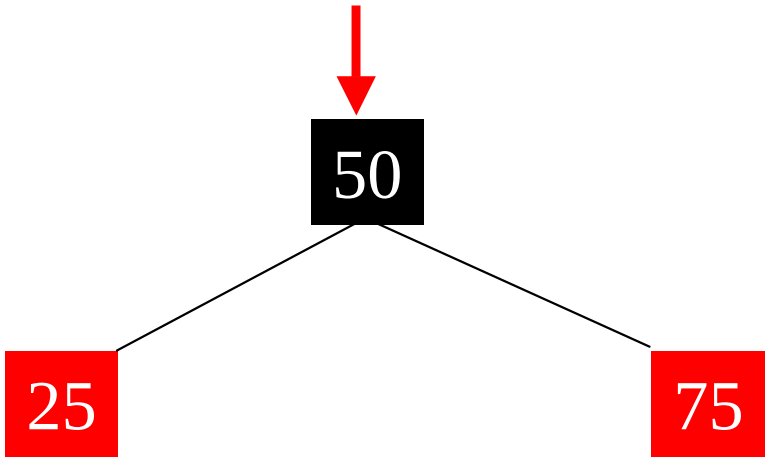
Result of further rotate left...



The tree again violates the red-black rules.

We would never do a rotation and leave it in this state.

Better rotate right again!



We’re back where we started!

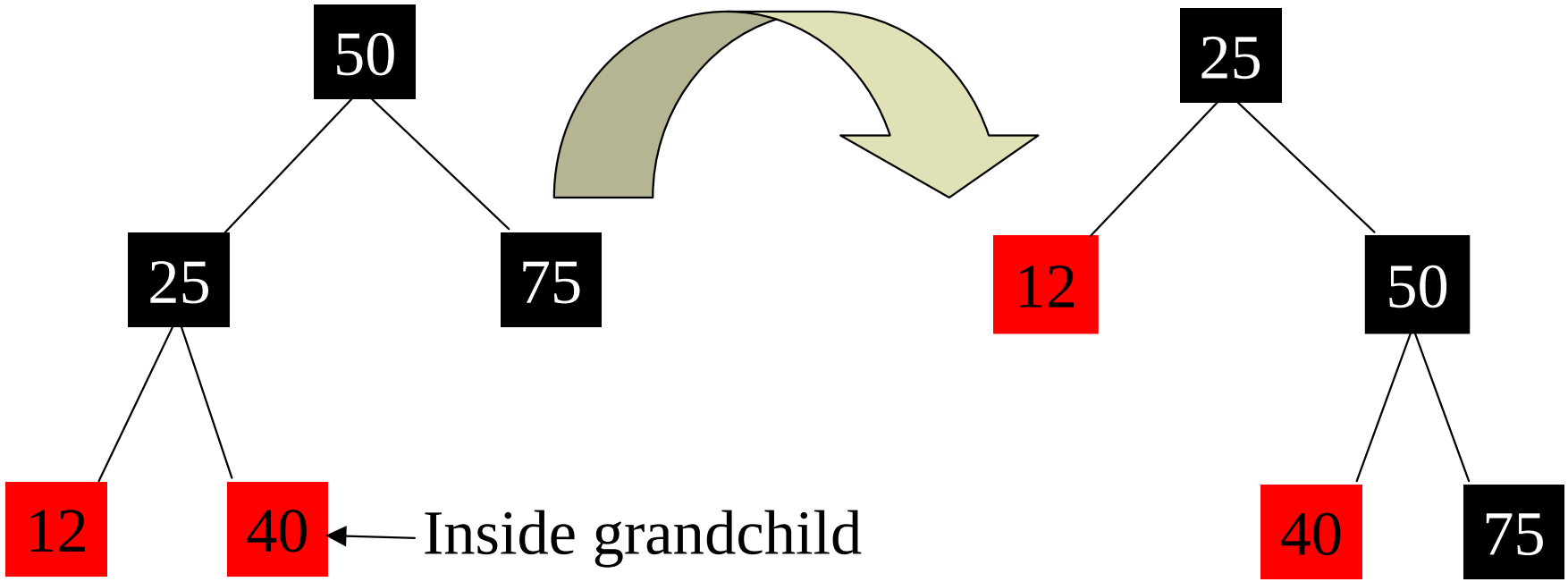
The tree is now red-black correct once again

In practice, results can be a bit more complicated than this, since no “grafting” operations were required in this example.

So what's grafting?

- Rotations take place with respect to the root of some sub-tree.
- If there is an inside grandchild, it is removed from its parent, and is reconnected to sub-tree root as it rotates down.
- This inside grandchild is called a crossover node, because it crosses over to the other sub-tree of the sub-tree root as it rotates down.

Right rotation about 50



50 (the root) rotates down and to the right

25 and 12 rotate up and to the right.

40 is disconnected from 25 and reconnected to 50

Result in this example is not red-black correct.

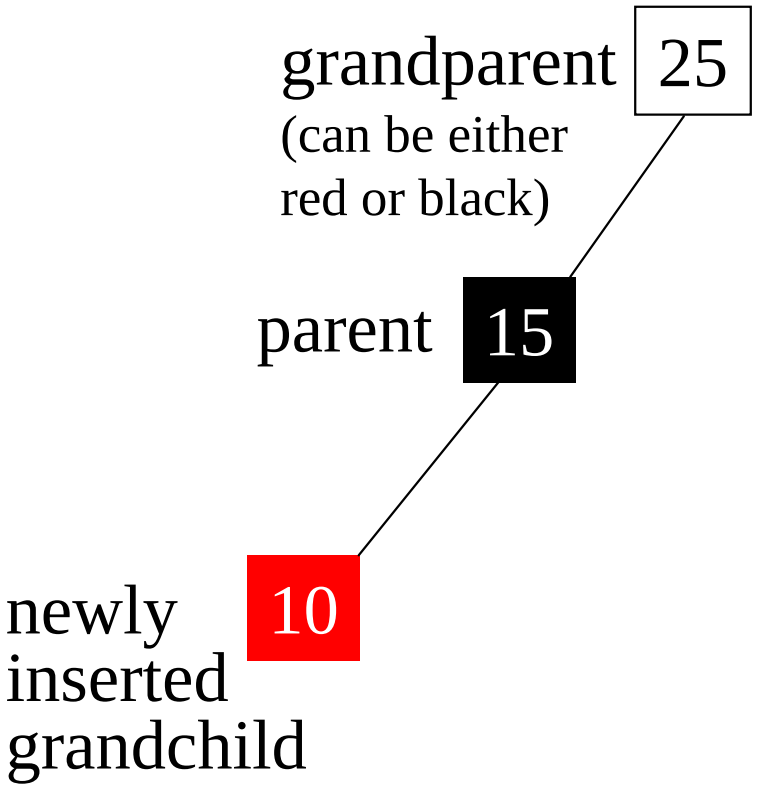
We're just trying to understand how rotations work for the time being, so don't worry about this now.

Rotations once a new node is inserted:

Three different cases exist:

- Parent of new red leaf node is black
- Parent is red and new red leaf node is an “outside” grandchild while “inside” grandchild is null.
- Parent is red and new red leaf node is “inside” while “outside” grandchild is null.

(Case 1) Parent is black...



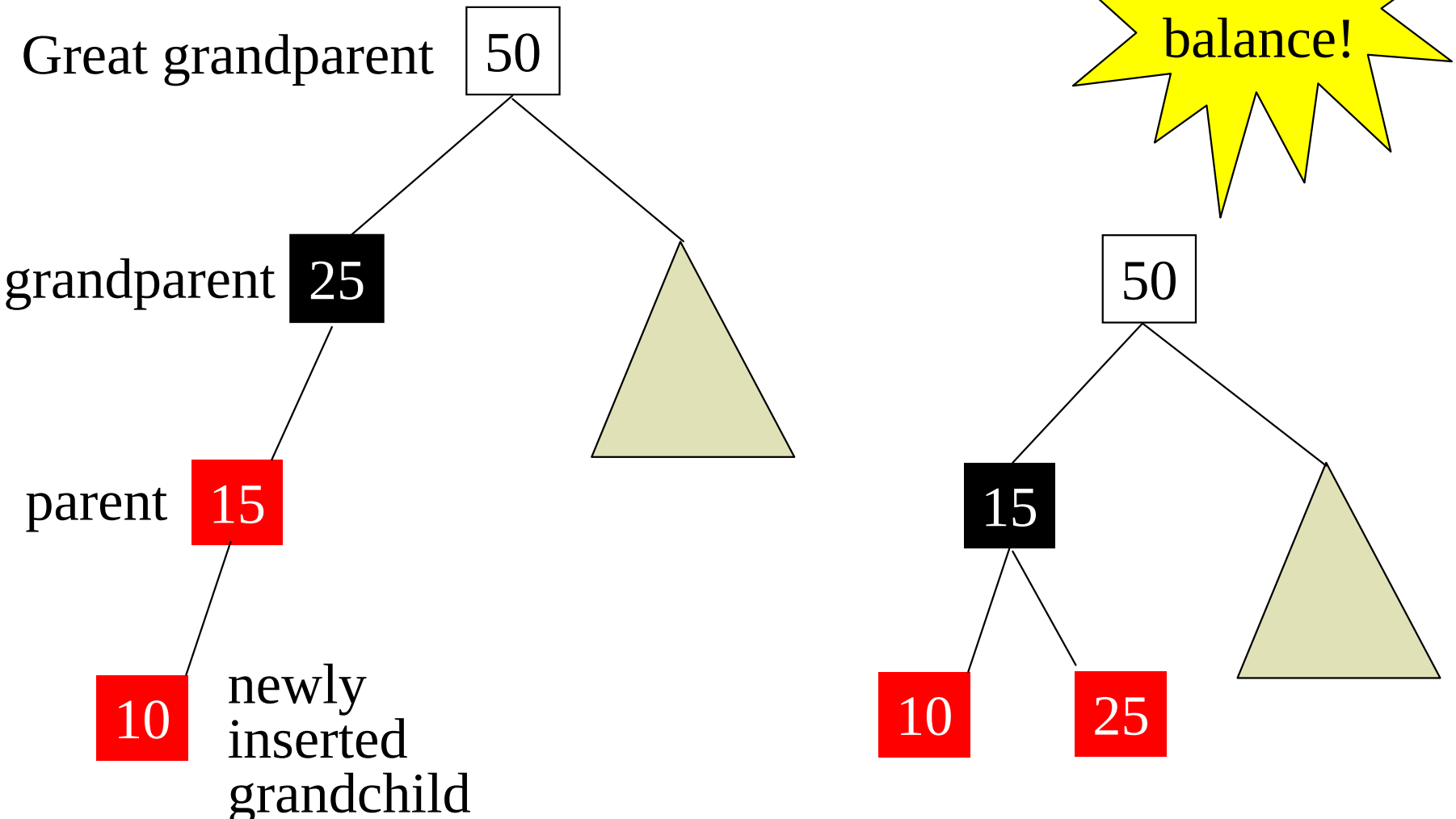
This case doesn’t violate black height rule since the inserted node is red.

No red-red conflict.

We’re done!

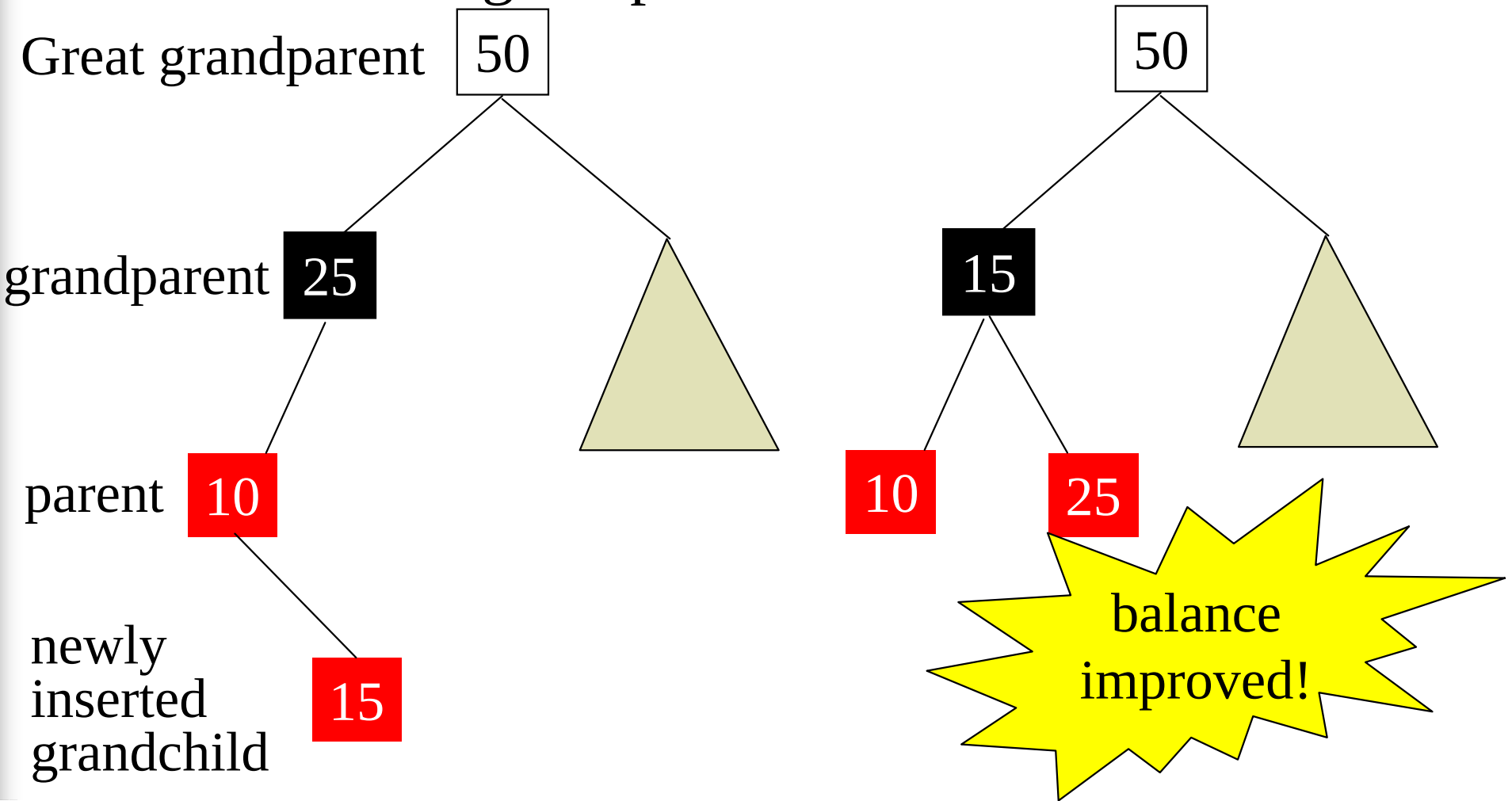
(Case 2) Parent red, new node is "outside"

- Switch colour of grandparent.
- Switch colour of parent
- Rotate grandparent down to raise outside node



(Case 3)Parent red, new node is "inside"

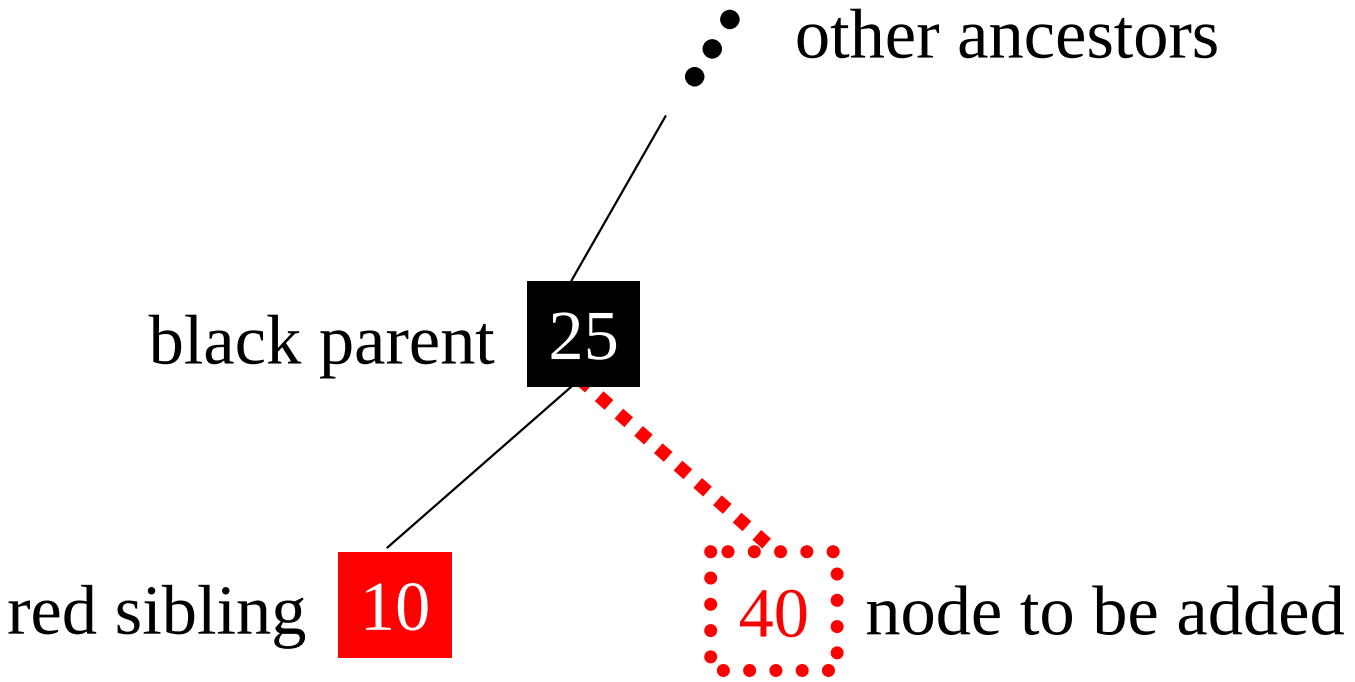
- Switch colour of grandparent.
- Switch colour of new node
- Rotate about parent to raise new node
- Rotate about grandparent to raise new node



Other cases we should consider

- New node has a sibling and parent is black
- New node has a sibling and the parent is red
- Black parent has a sibling
- Red parent has a sibling

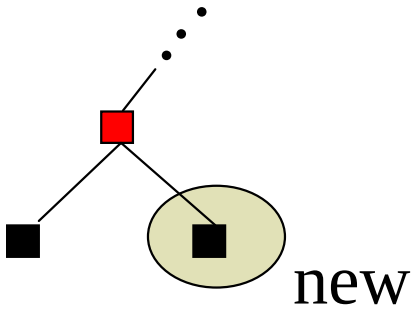
Node has a sibling and black parent



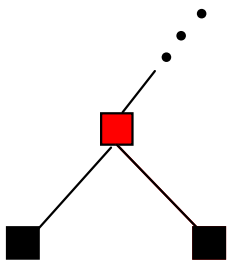
If the parent is black, then sibling of new node must be red, otherwise black height rule is violated in the original tree!

In this case, the (red) new node can be inserted with no changes, flips or rotations!

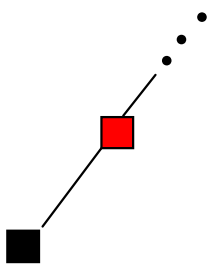
Node has sibling, parent is red



Post-insertion, if parent is red, then both siblings must be black, given Red Parent Rule.



While we might insert the new red node and flip it to black, this is actually an impossible scenario. Why?

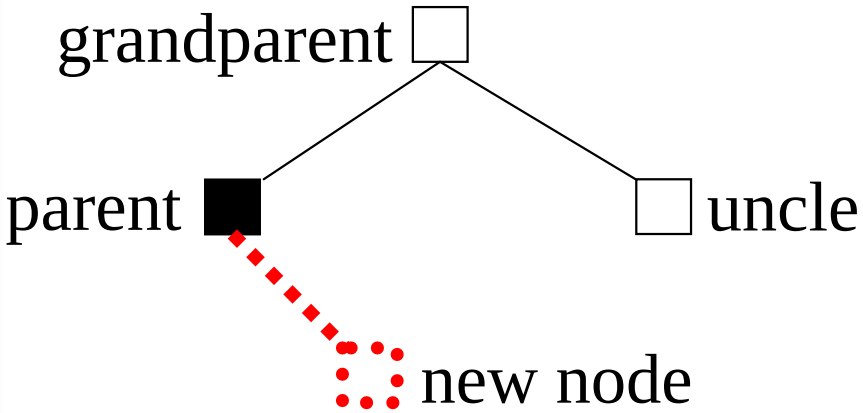


If the future sibling is black, it violates the the Black Height Rule since there's a different black height through the parent's right null node.

Resort to Rotation!

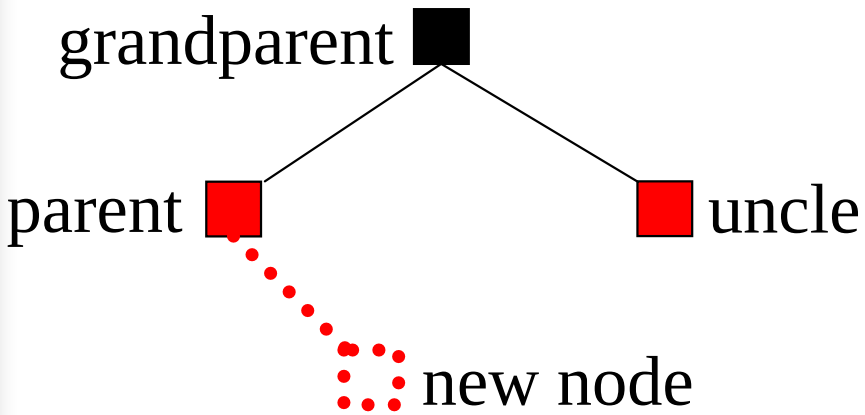
- See later slides

Black parent has a sibling



If parent is black, can insert new node with no problems.

Red parent has a sibling



If parent is red, then uncle is red and grandparent black . This case doesn’t arise for insertion, however, since black nodes with two red children are flipped on the way down!

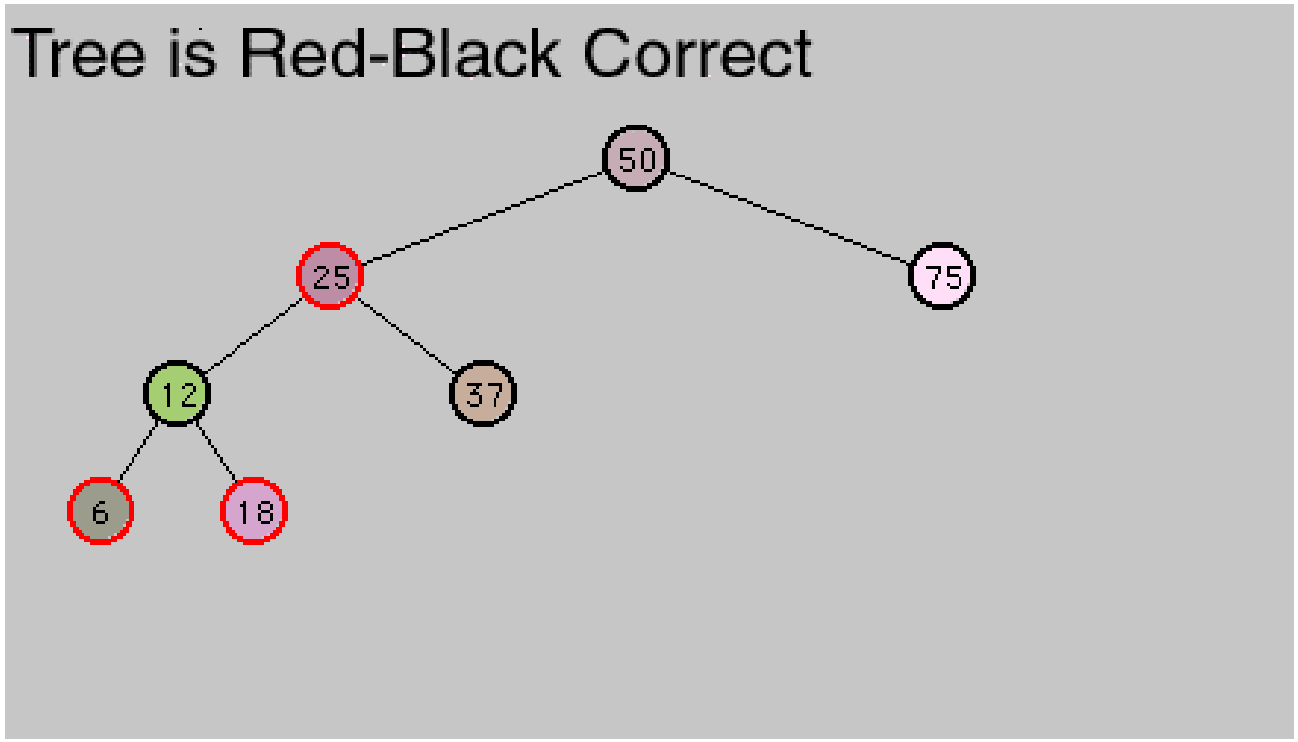
Flips can lead to Red Parent violations

- If a black node with two red children is encountered, it's flipped on the way down.
- Colour flips on the way down can cause a violation of the Red Parent rule.
- This can be fixed with a rotation.
- Two rotation scenarios are when the violating node is an outside grandchild or when its an inside grandchild.

Fixing Red-Red violations after flip when violator is an outside grandchild (see slide 34)

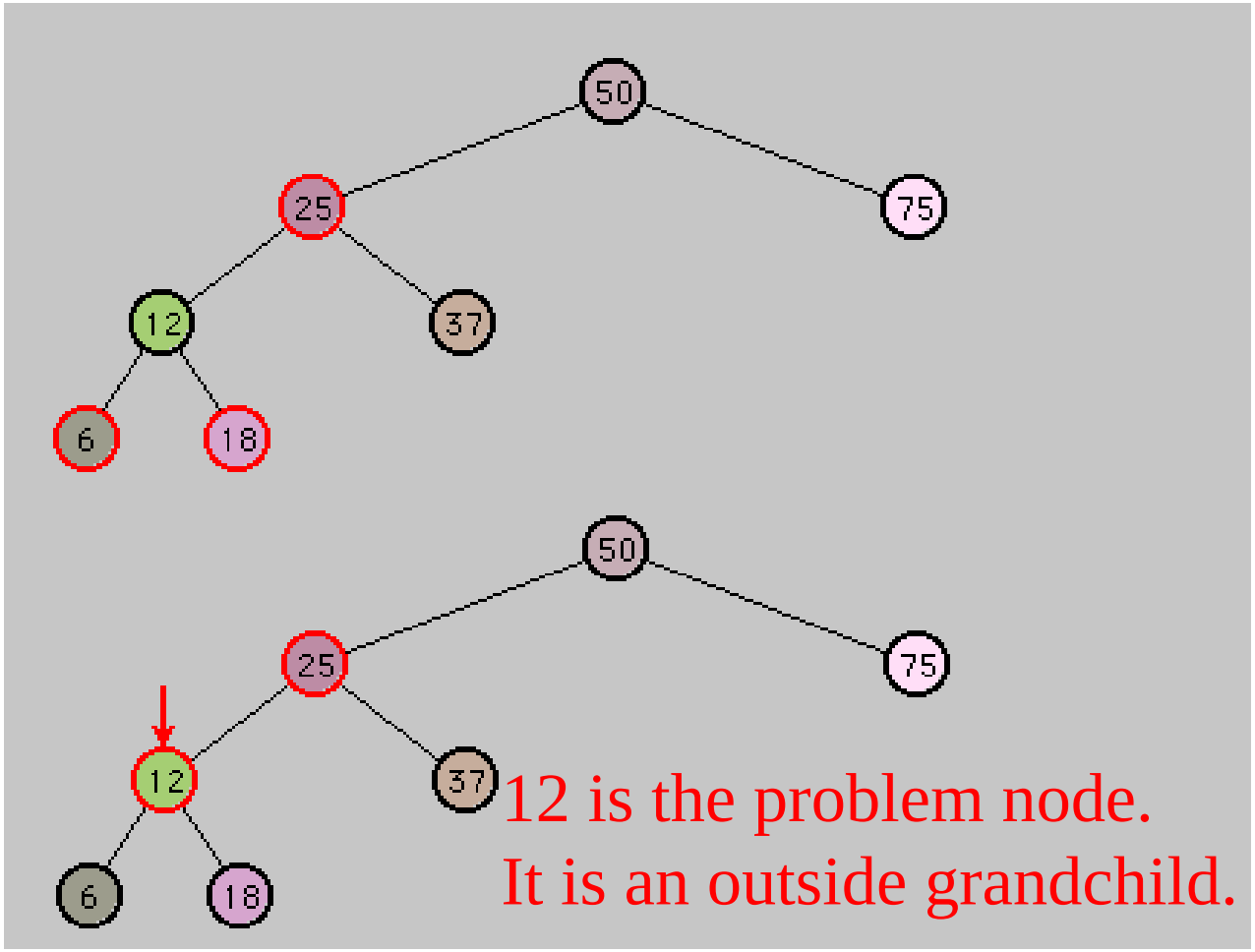
- Switch colour of violator's grandparent
- Switch colour of violator's parent
- Rotate about grandparent in the direction that raises the violating node upwards.

Example...



Created using LaFore's RBTree Workshop Applet
Red nodes are shown with a red outline.
Black nodes are shown with a black outline.
Verify that this tree is Red-Black Correct.

Inserting 3: Flips on the way

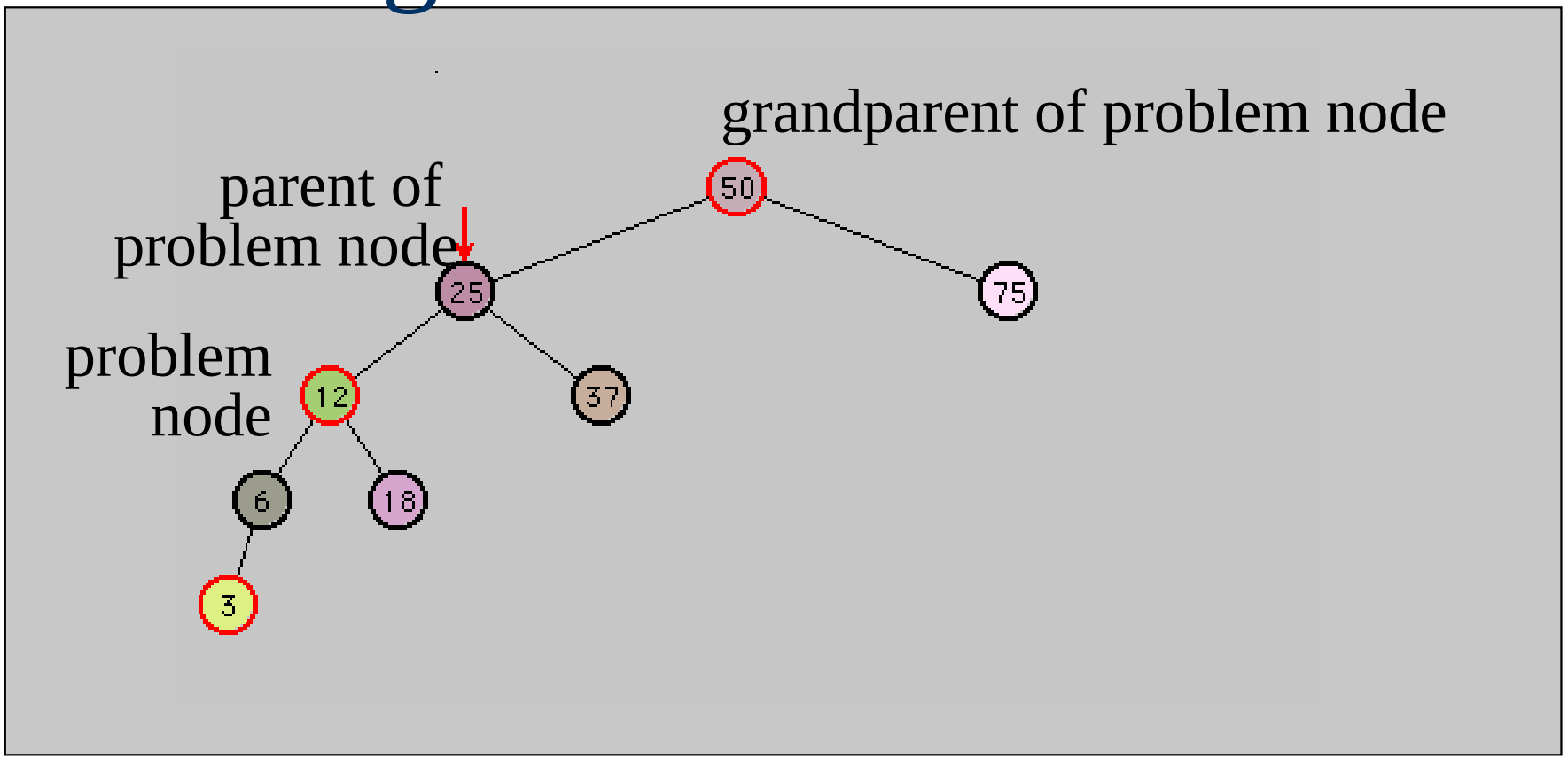


Eventually, 3 will be a left child of 6

On way to the insertion point, flip if parent is black and children are both red, as in nodes 12, 6, and 18

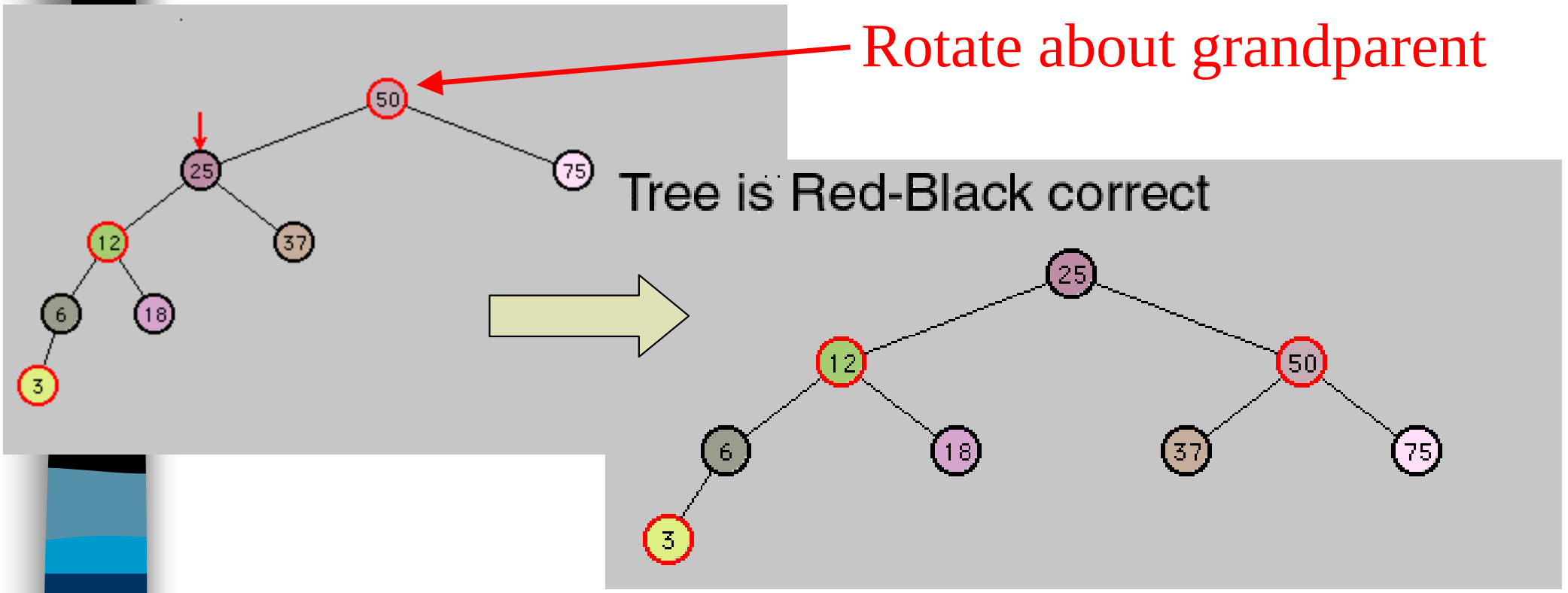
Produces Red-Parent rule violation at node 12.

Inserting 3: Before rotation



Parent and grandparent have had colour switch, but still needs a rotation

Insert 3: After rotation



Rotate in a clockwise direction to demote 50 and promote 25 and 12.

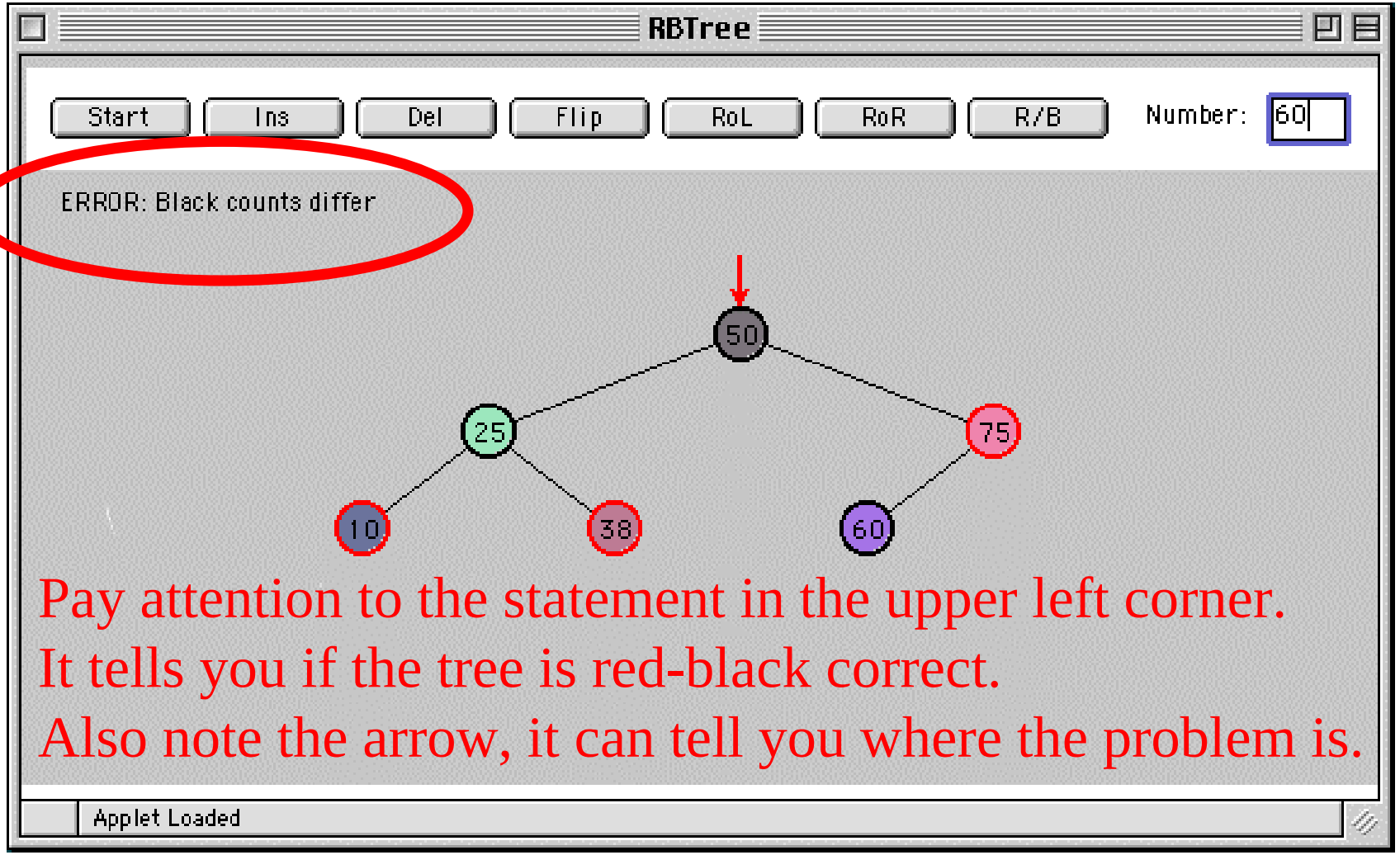
37 is removed from 12 and grafted to 50.

After rotation, tree is well balanced

Rules for an inside grandchild problem (see slide 35)

- Switch colour of problem node.
- Switch colour of grandparent.
- Rotate from parent in direction that raises problem node.
- Rotate from grandparent in the direction that raises the problem node.

LaFore's RBTree Workshop Applet



Pay attention to the statement in the upper left corner.
It tells you if the tree is red-black correct.
Also note the arrow, it can tell you where the problem is.

\DSA\JAVAAPPS\JAVA-1.1\Chap09\RBTree\RBTree.html

LaFore's RBTree menu

- **Click** on node to move arrow to it
- **Start** makes a new tree with one node
- **Ins** inserts a new node with value N in Number box
- **Del** deletes the node with value N in Number box
- **Flip** swaps colours between black parent (arrow) and two red children
- **RoL** rotates left around node with arrow
- **RoR** rotates right around node with arrow
- **R/B** toggles colour of node with arrow

Example

- Insert 8 10 6 3 5 2 1 into a BST

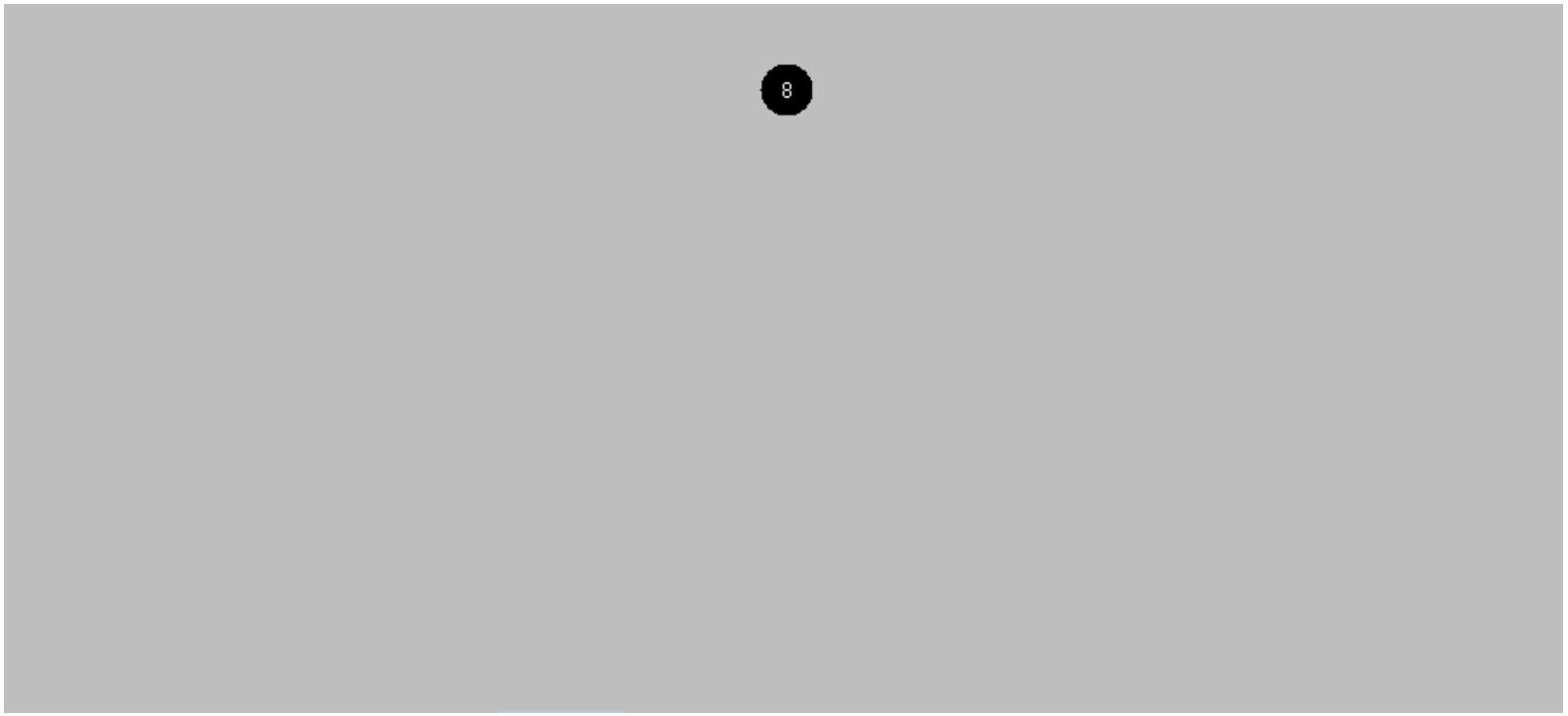
You try

Example

- Now insert 8 10 6 3 5 2 1 into a RB Tree

Example

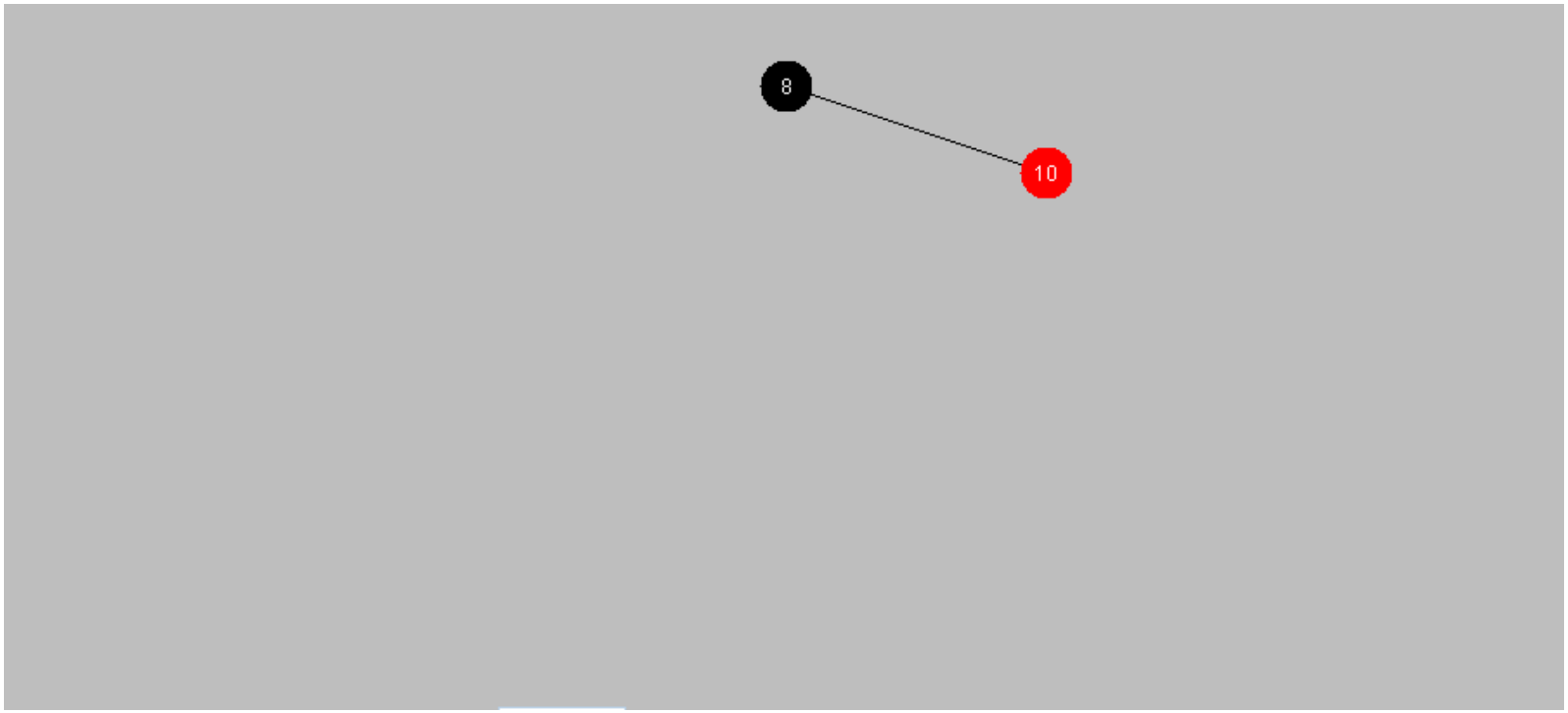
- Insert 8 10 6 3 5 2 1 into a RB Tree



8 - Black root added

Example

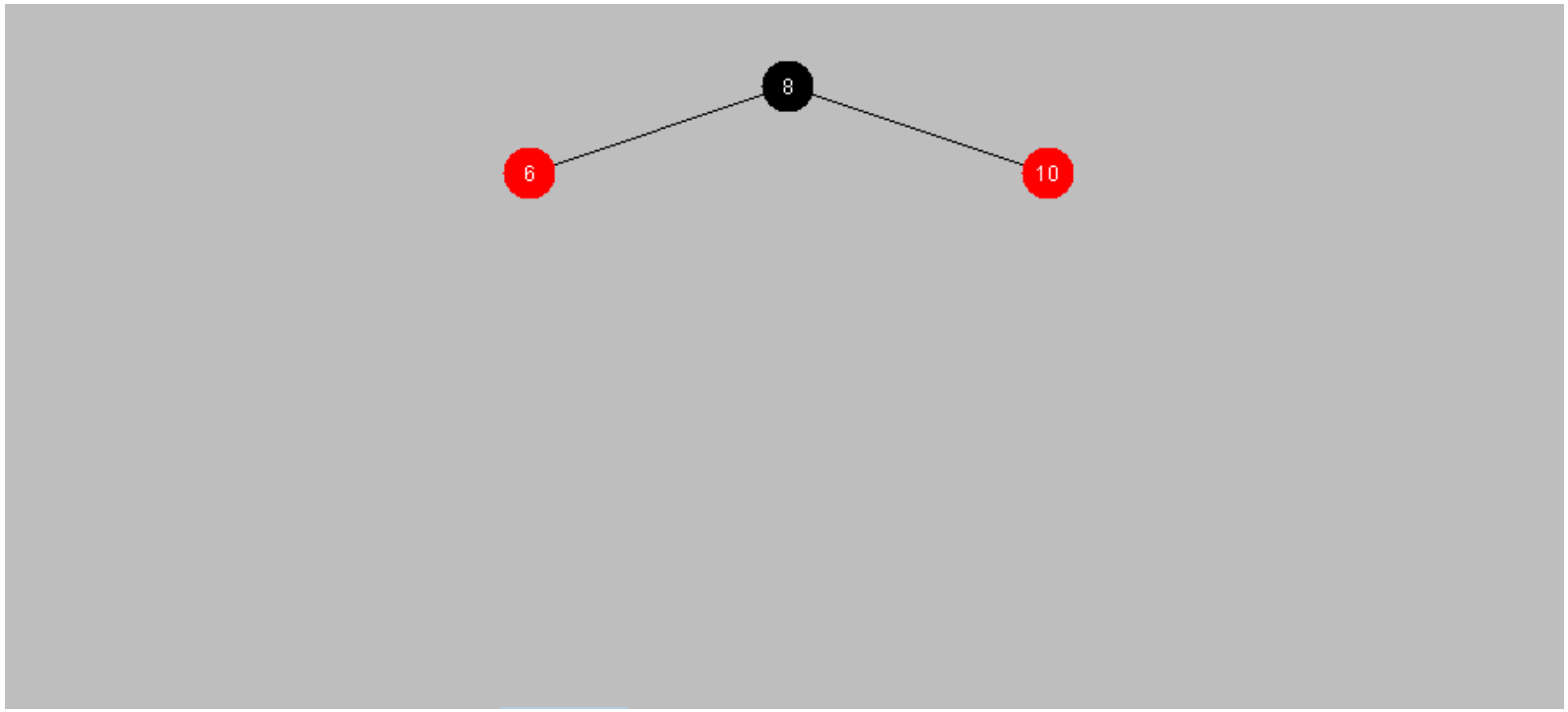
- Insert 8 10 6 3 5 2 1 into a RB Tree



10 – Red node added. No change.

Example

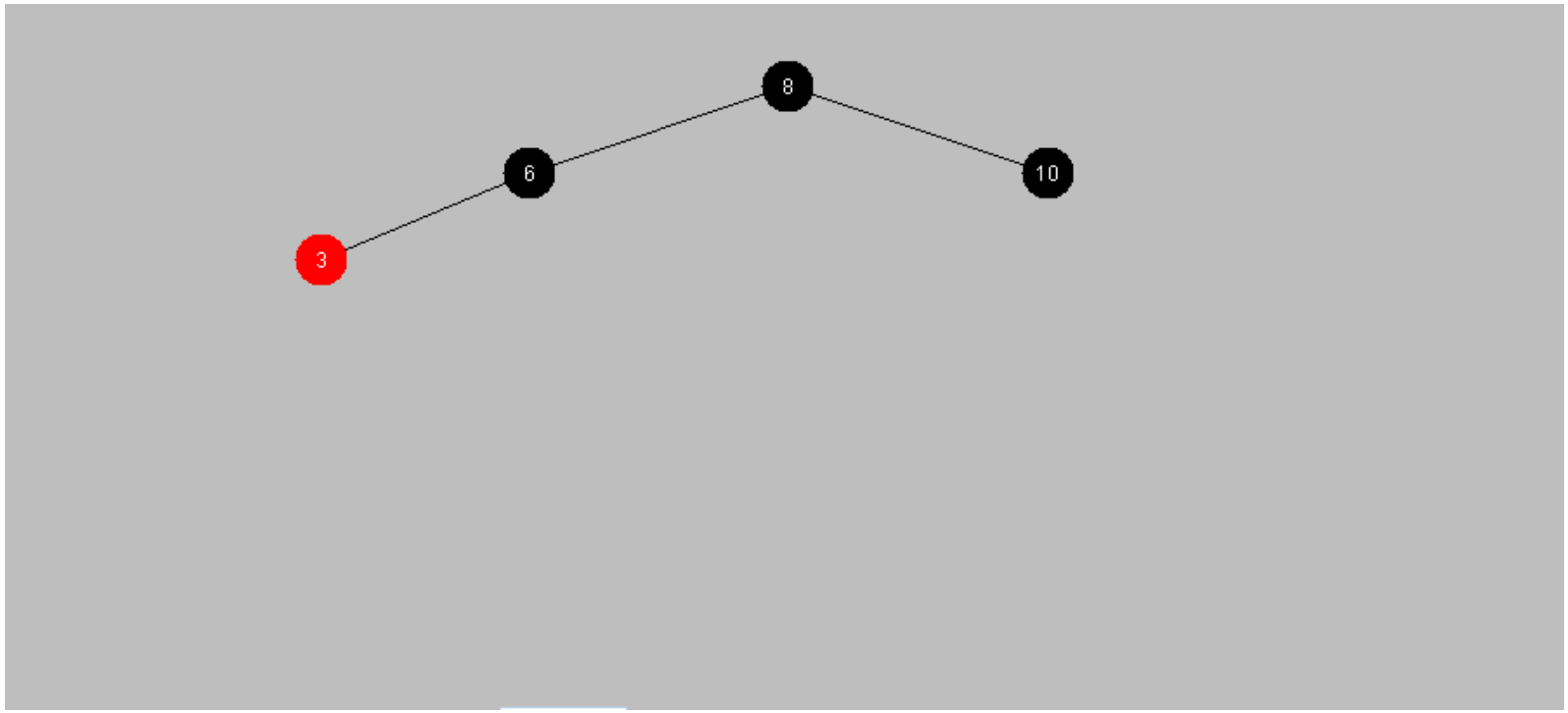
- Insert 8 10 6 3 5 2 1 into a RB Tree



6 – Red node added. No change.

Example

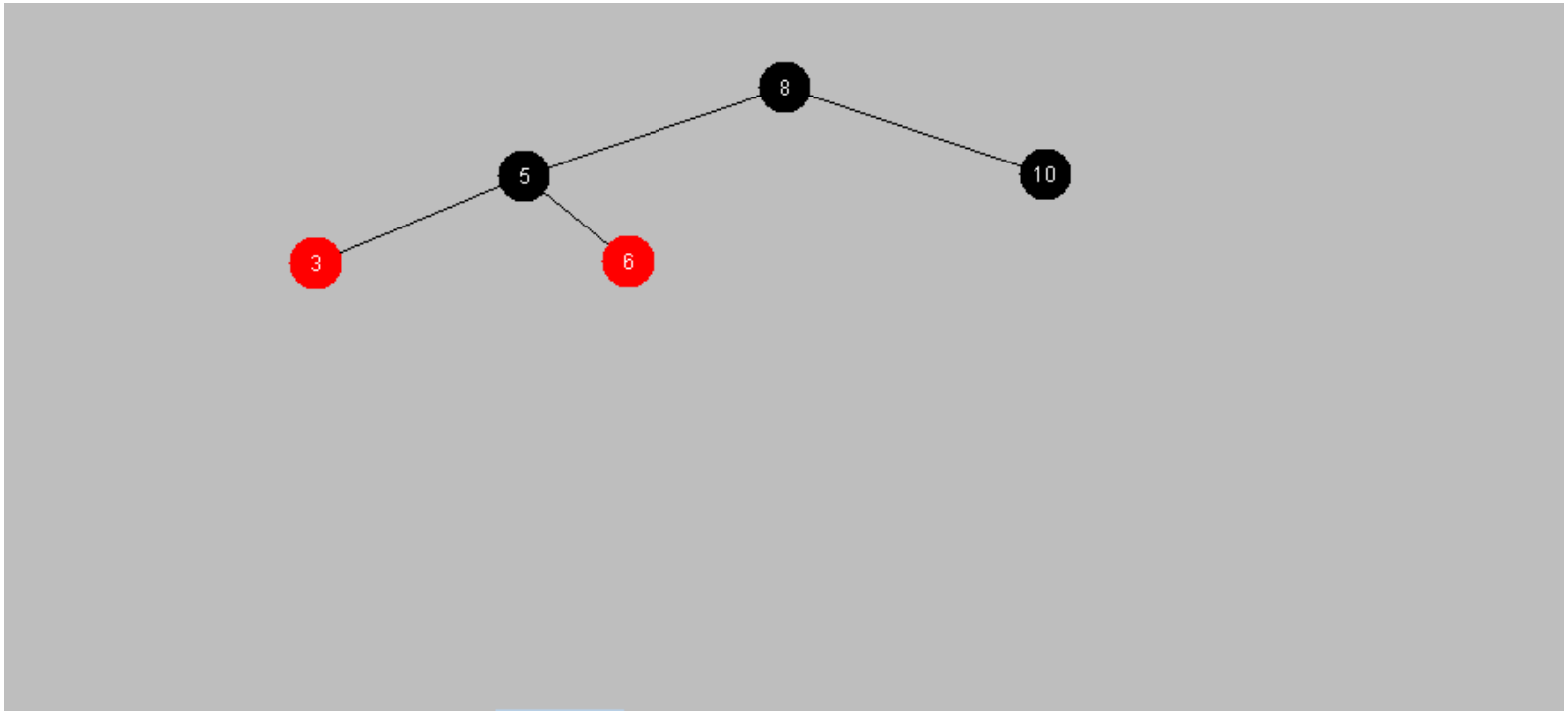
- Insert 8 10 6 3 5 2 1 into a RB Tree



3 – (1) Flip root and children. (2) Switch colour of root.
(3) Insert 3 (red) node

Example

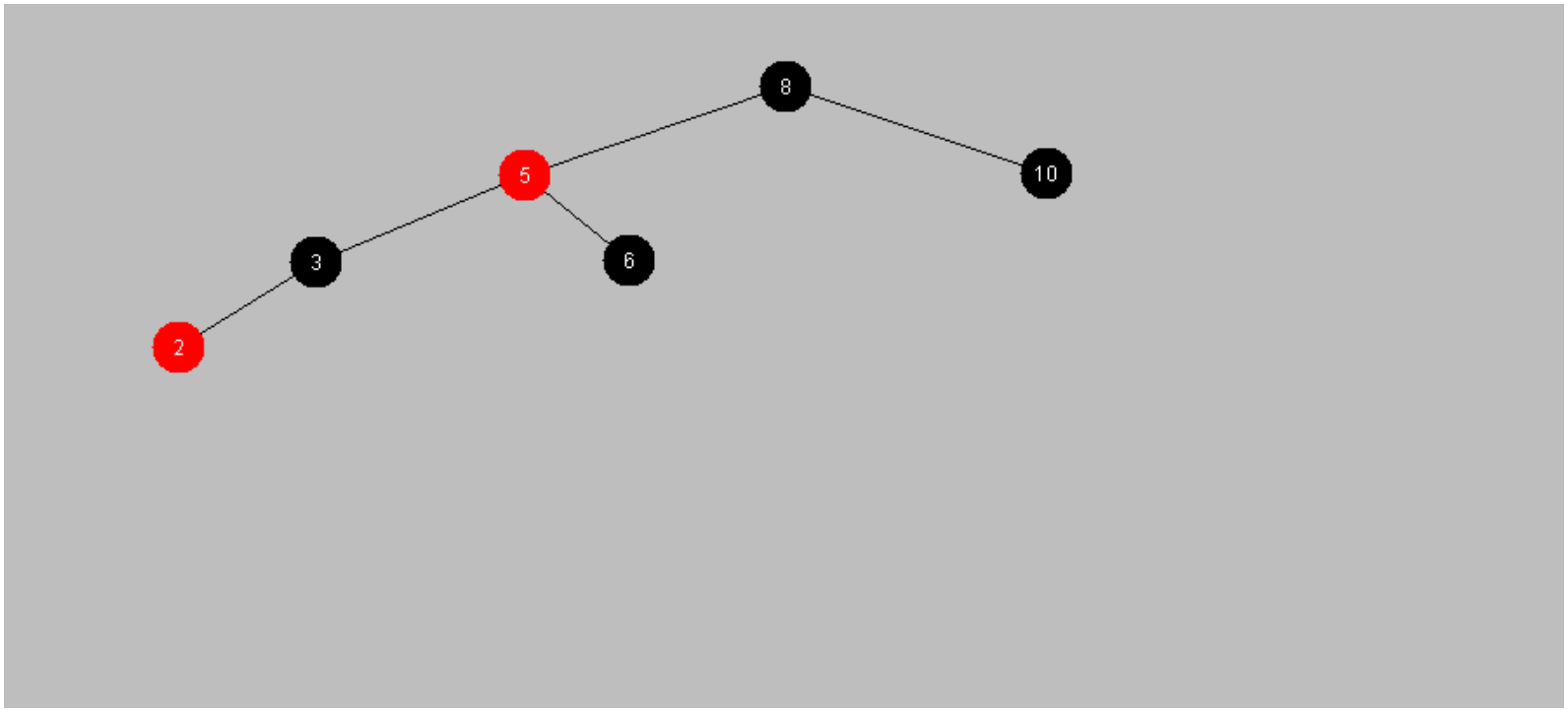
- Insert 8 10 6 3 5 2 1 into a RB Tree



5 – Parent is red new node is inside. Solution: (1) Switch colour of Grandparent
(2) Switch colour of new node. (3) Rotate around Parent. (4) Rotate around Grandparent.

Example

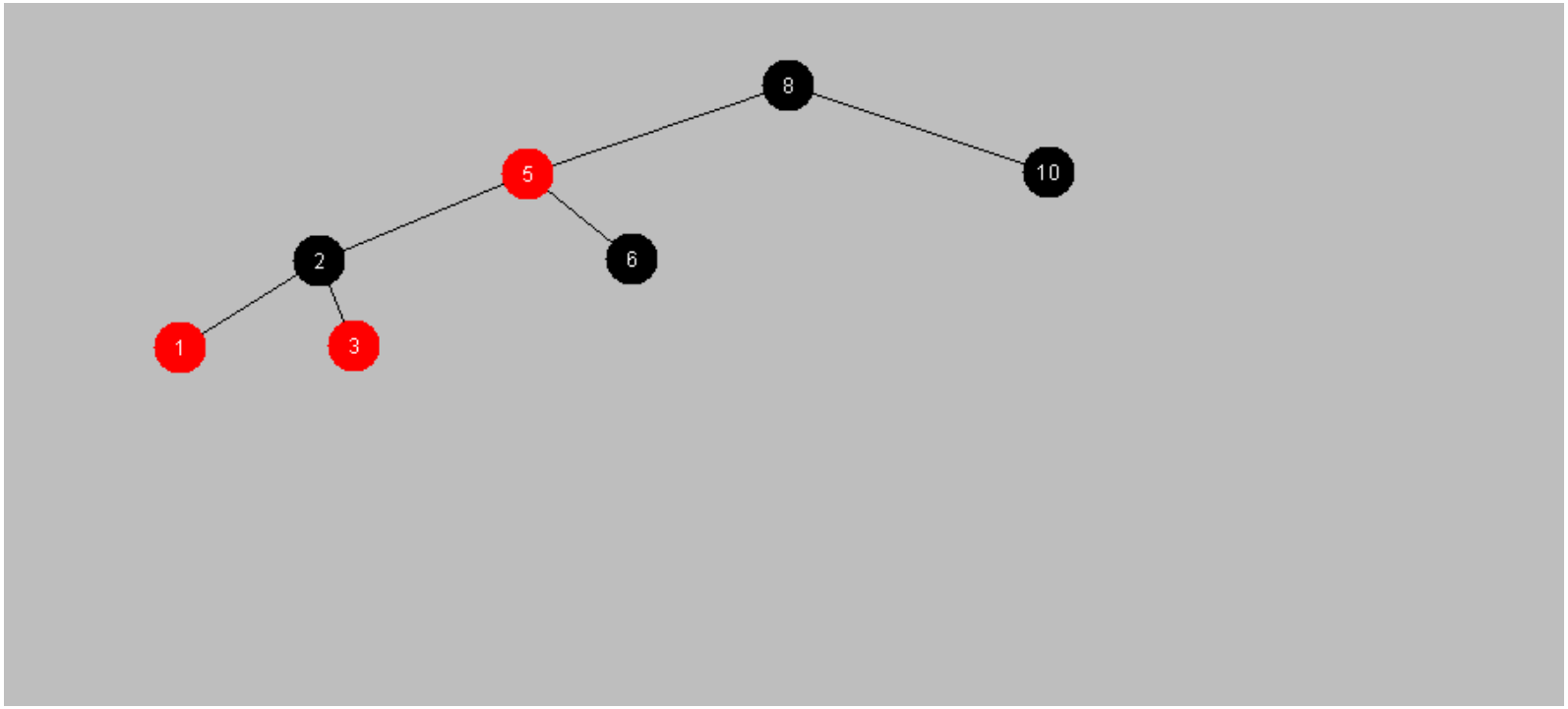
- Insert 8 10 6 3 5 2 1 into a RB Tree



2 – (1) Flip node 5 and children. (2) Insert 2 (red) node

Example

- Insert 8 10 6 3 5 2 1 into a RB Tree



1 – Parent is red new node is outside. Solution: (1) Switch colour of Grandparent (2) Switch colour of parent. (3) Rotate around Grandparent.

Why all the fuss?

- Important! Use LaFore's RBTree Workshop Applet to prove to yourself that it is impossible to create a red-black correct tree that is unbalanced by more than two levels!

Summary of Steps for insertion

1. Node to be inserted is set to red
2. Start at root node
3. If parent is black and children are both red, flip colors.
4. If flip violates RB property, then perform rotations:
   - For outside grandchild:
     - Switch colour of grandparent. Switch colour of parent. Rotate grandparent down to raise problem node
   - For inside grandchild
     - Switch colour of grandparent. Switch colour of problem node. Rotate about parent to raise problem node. Rotate about grandparent to raise problem node
5. Process continue until leaf node

Conclusion

- RBTree use to ensure binary tree generated is close to balance.
- Understanding of the RBTree rules and how they are used to generate a binary tree.
- You are not required to implement a RBTree in this course.
- Important to know how to use these rules to generate a RBTree.