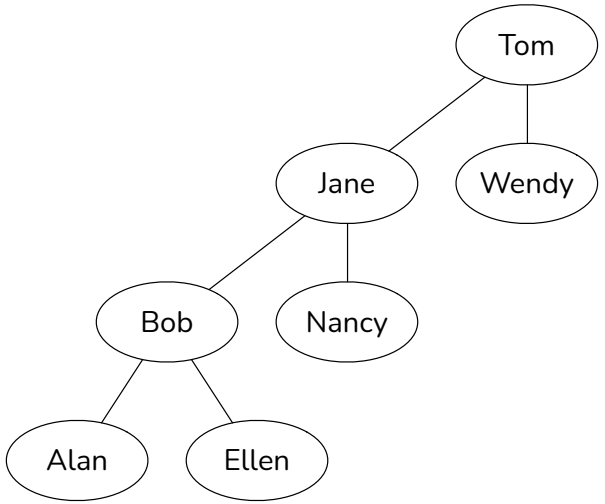
Red-Black Trees

Kristian Guillaumier

Binary Trees

- One way to index records using a search key is to use binary trees.
- If the tree is balanced then we can perform very efficient inserts, deletes, and lookups (binary search tree).
- However, the ‘basic’ binary tree can degenerate.



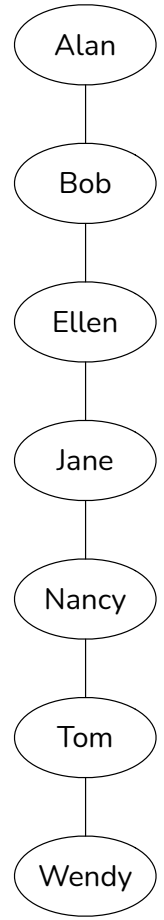
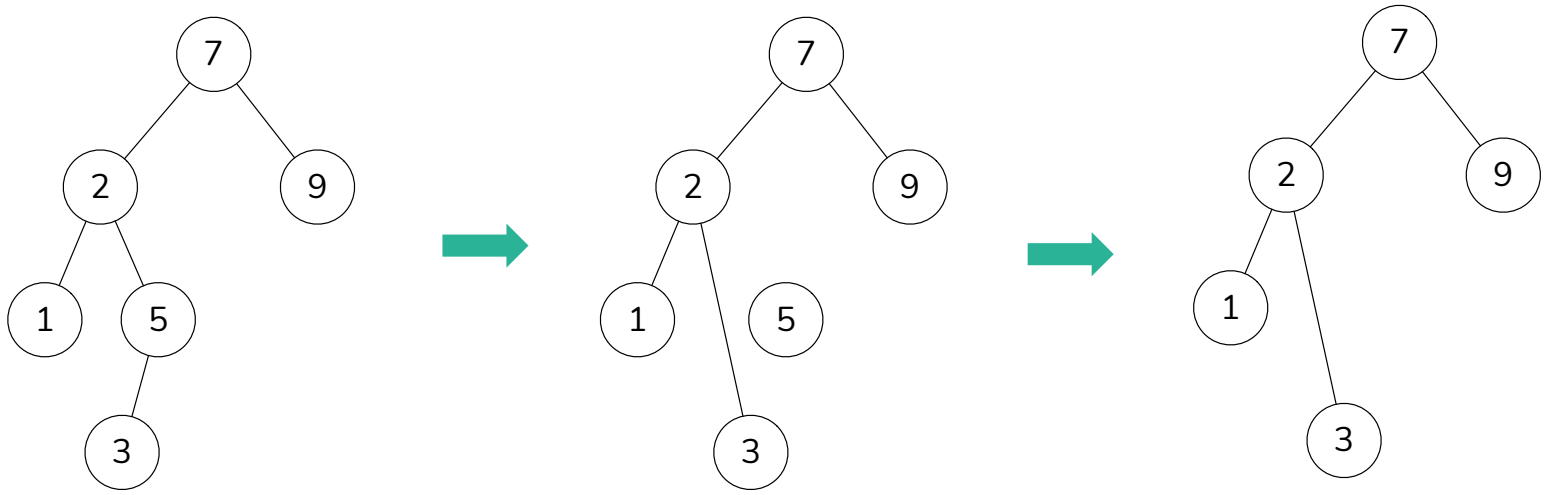


Image from Weiss

Some useful operations on binary trees

- Find minimum/maximum: trivial – start at root and branch left/right until there is no child.
- Find: recursive left/right decisions. Insert is a variation on find.
- Delete: most complex because it can disconnect the tree.
  - Scenario 1: Item is a leaf node – simply remove it since it will not disconnect the tree.
  - Scenario 2: Item X has only one child Y – set parent of X to point to Y and delete X…

Delete node 5:

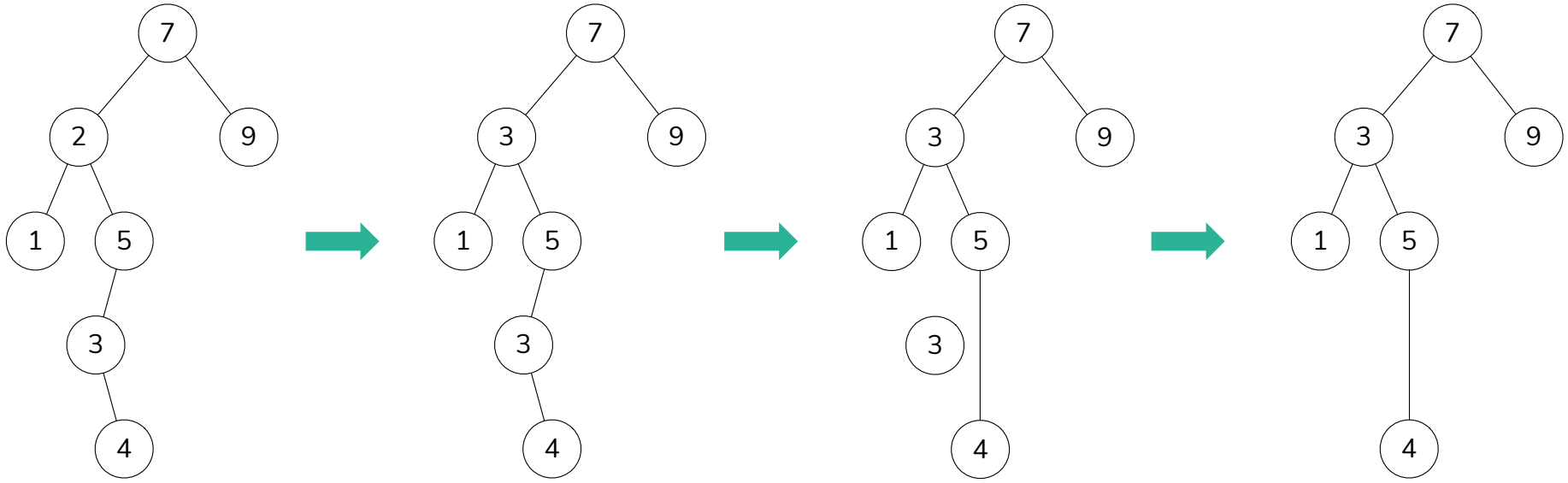


Example adapted from Weiss

...continued

- Delete scenario 3: node X has 2 children T^L and T^R,
  - Replace the item X with the smallest item Y in T^R.
  - The smallest item in T^R can be found efficiently.
  - Remove Y from T^R. Since Y in T^R can never have a left child, this will be simple.

Delete node 2:



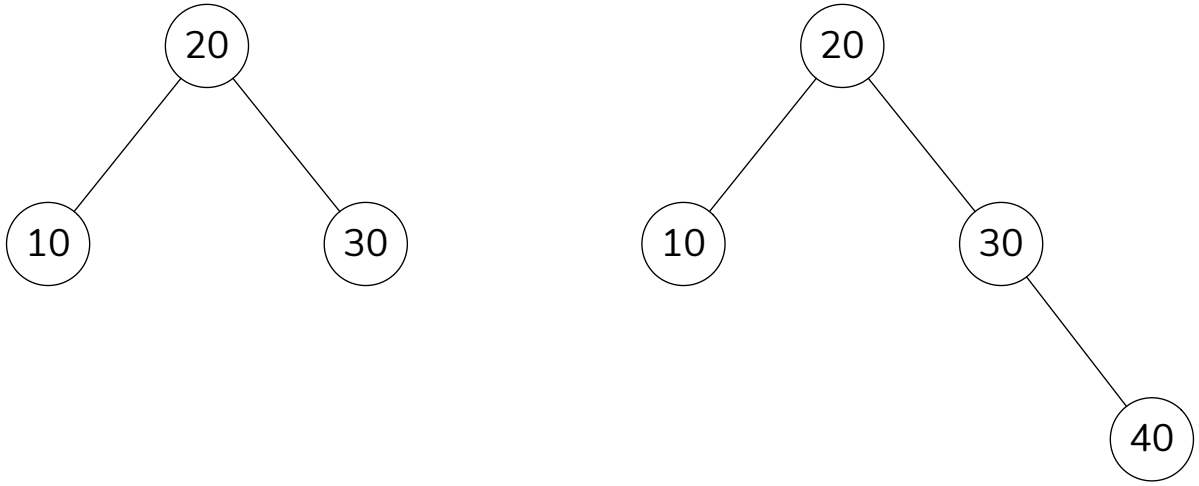
Example adapted from Weiss

Rotations in trees

- Recall AVL trees which maintain a height close to the minimum with little overhead.
- For any node in the tree the height of the left and right subtrees can differ by at most 1. The height of an empty tree is -1.
- After inserts or deletes the ‘shape’ of the tree is monitored to determine whether the AVL conditions have been violated.
- The height of the tree is ‘repaired’ using rotation operations.
- Note that not all inserts or deletes will necessarily cause a rotation to occur.

...continued

- Consider adding the value 40:



The tree is still balanced. No need to rotate.

...continued

- Consider adding the value 60:

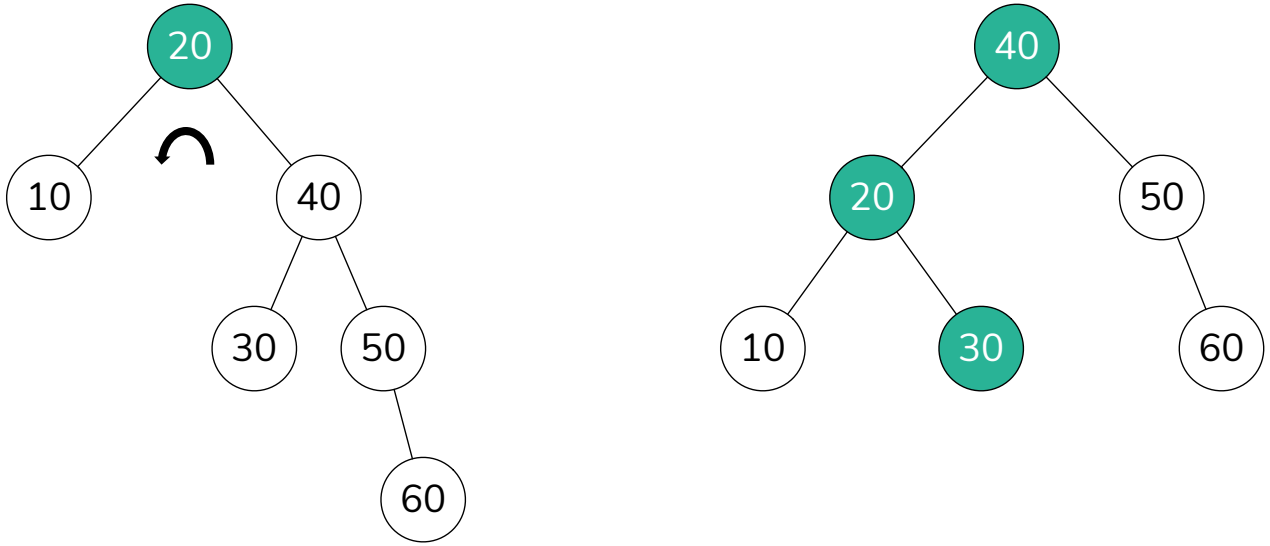


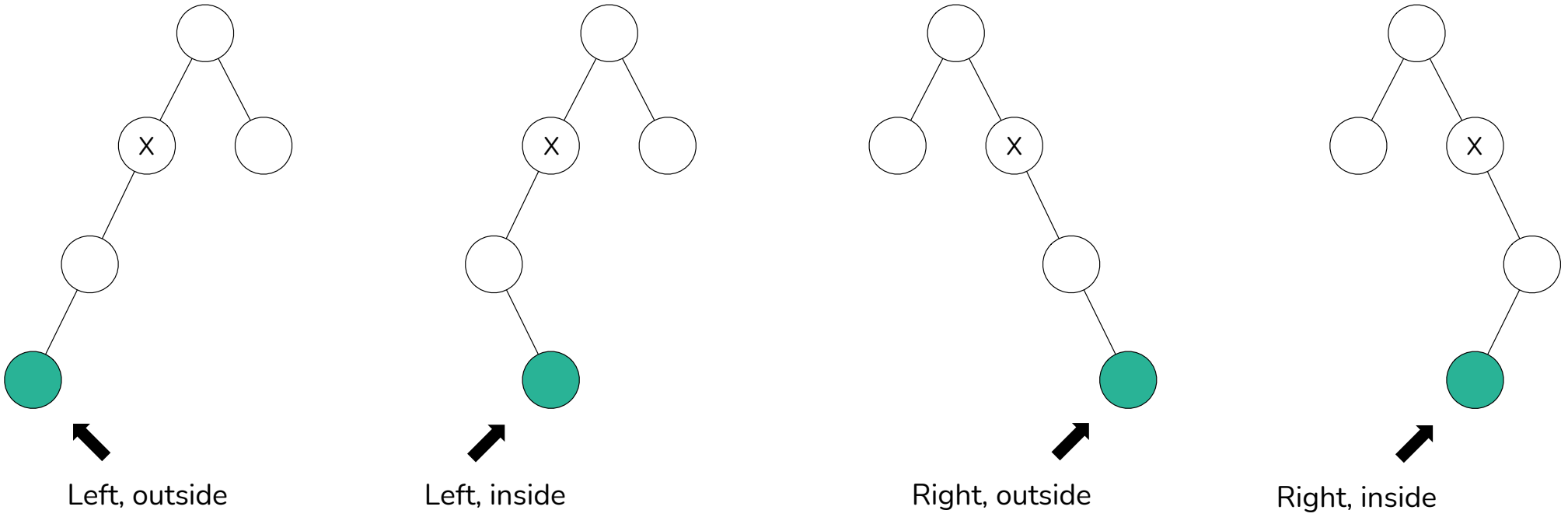
Image from Helman et al.

...continued

- We observe that after an insertion only a node on the path to the root might have the AVL property violated.
- We ‘walk up’ the tree from the insertion point until we find the node X that violates the AVL condition.
- The violation will occur in the following cases:
    1. Insertion in left subtree of left child of X.
    2. Insertion in right subtree of left child of X.
    3. Insertion in left subtree of right child of X.
    4. Insertion on right subtree of right child of X.

...continued

- Violations and notation:



...continued

- Example of LL rotation after inserting 1 (RR is the same by symmetry):

```
RotateLL(Node n) → Node {
    Node k  = n.Left
    n.Left  = k.Right
    k.Right = n
    return k
}
```

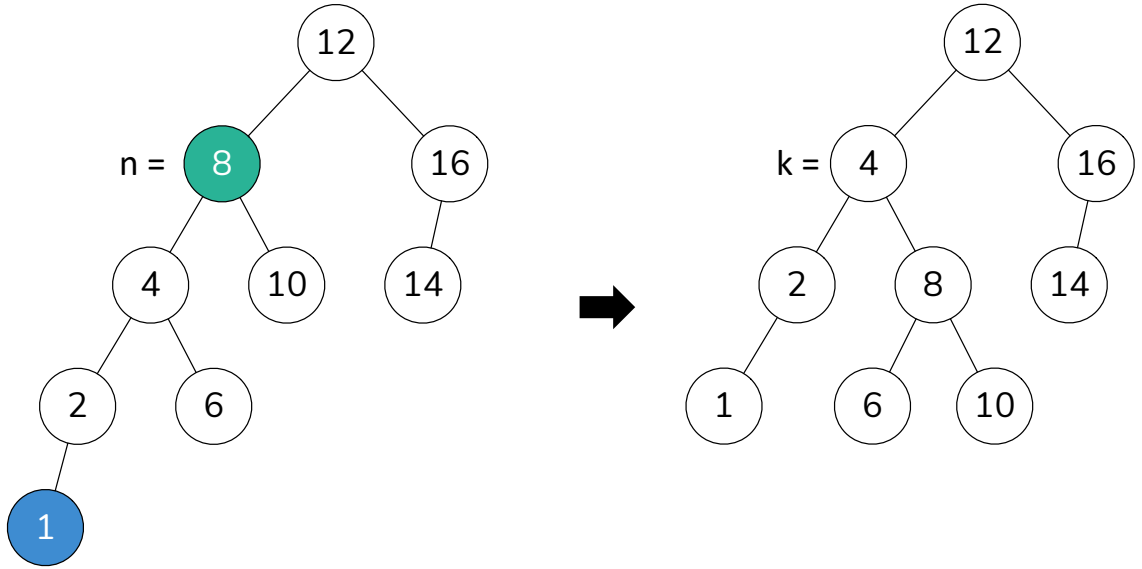
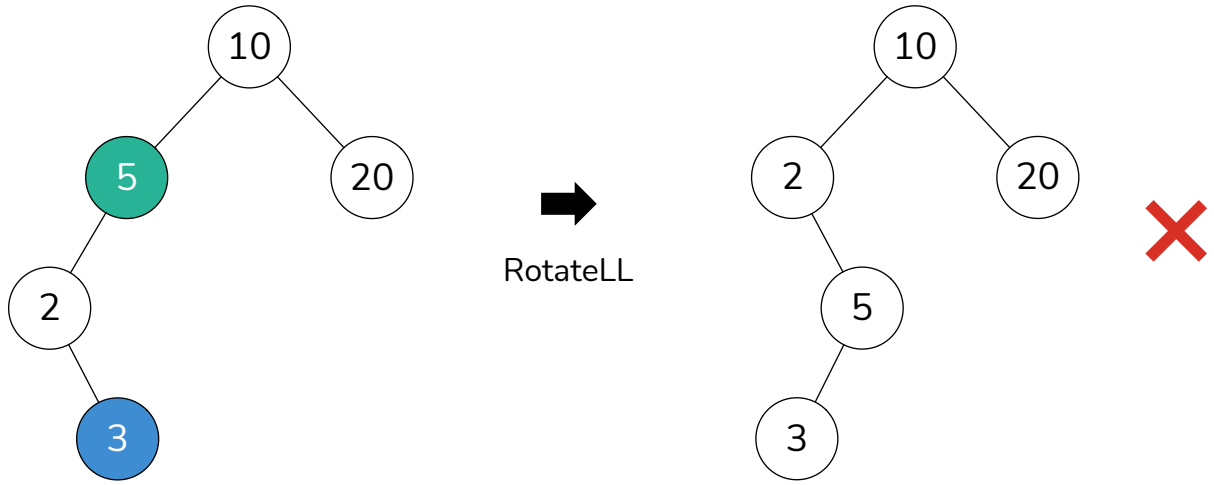


Image from Helman et al.

...continued

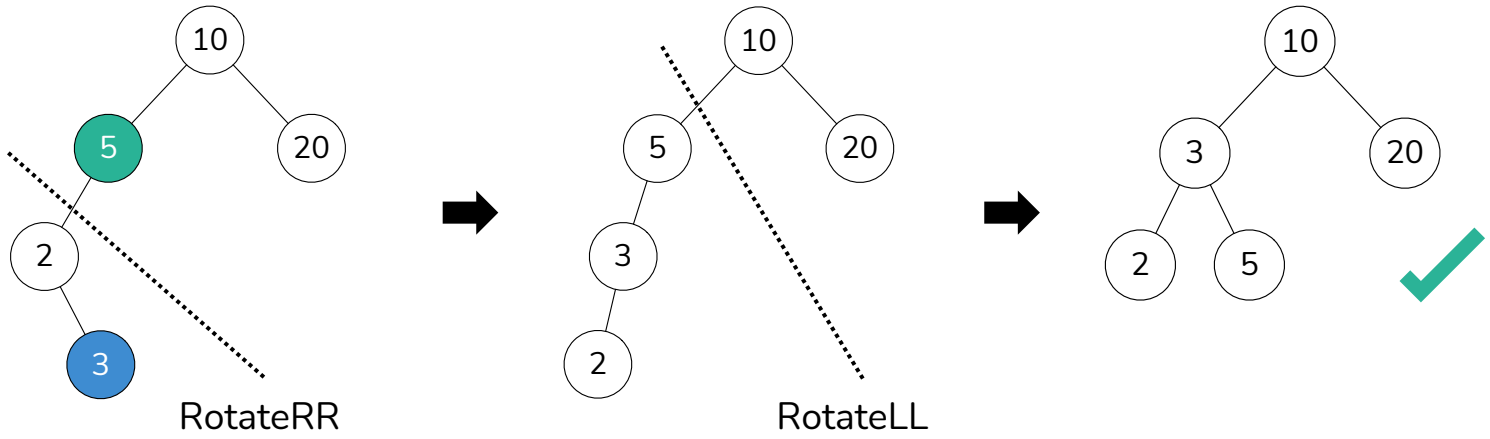
- RotateLL and RotateRR takes care of 'left and right outside' cases respectively, but not 'left and right inside' cases.
- Consider what happens if we execute RotateLL on a 'left, inside' case:



...continued

- The solution is to use double rotation:

```
RotateLR(Node n) → Node {
    n.Left = RotateRR(n.Left)
    return RotateLL(n)
}
```



Red-black trees (RBTs)

- A simple AVL tree implementation needs to:
  - Recursively go down the tree to find the insertion point.
  - Then go up to find the height information and rebalance (during recursive unwinding).
- RBTs can be implemented iteratively using single top-down pass in a relatively simple way. RBTs also tend to require fewer rotation operations during insertion and deletion.

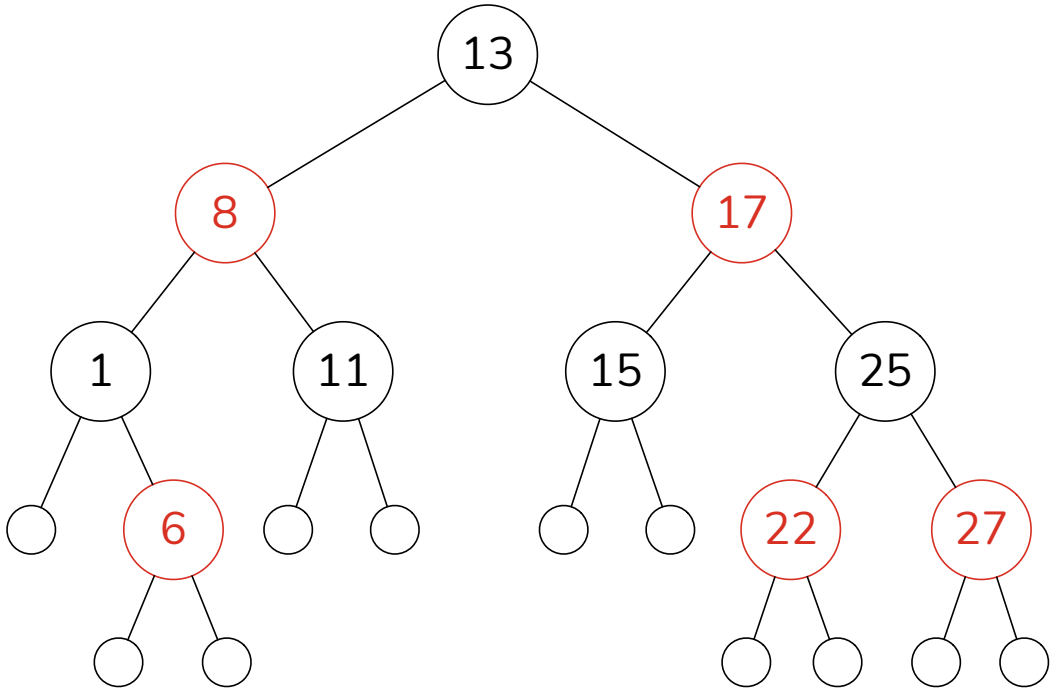
Properties of RBTs

A red-black tree is a self-balancing binary tree.

1. Nodes are coloured red or black.
2. The root is black.
3. Children of a red node are always black (there can never be two consecutive red nodes in a path).
4. Every path from any node to the null leaves has the same number of black nodes.
5. Null leaves are black.

An example

1. Nodes are coloured red or black.
2. The root is black.
3. Children of a red node are always black (there can never be two consecutive red nodes in a path).
4. Every path from any node to the null leaves has the same number of black nodes.
5. Null leaves are black.



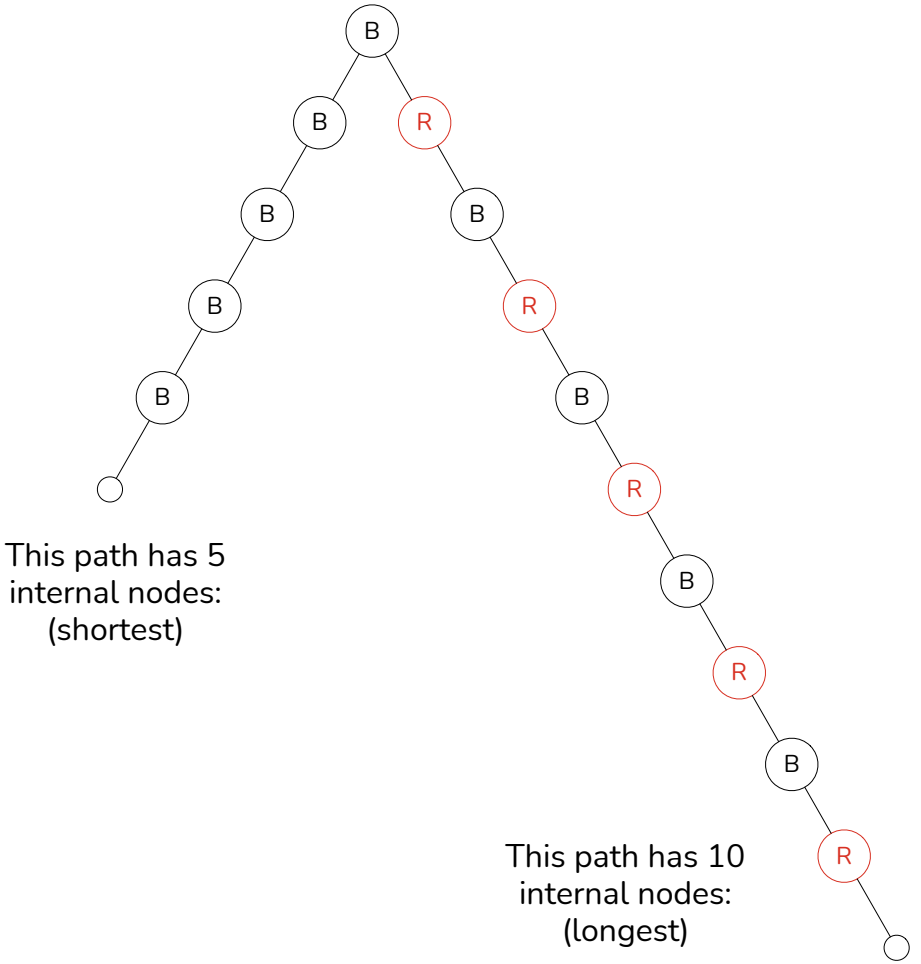
Note that the RBT properties do not specify that it must also be a BST – this example just happens to be a BST as well.

Image adapted from Wikipedia

A very important observation

- The shortest path will consist of only black nodes.
- The longest path will consist of alternating black and red nodes. This is because there can never be two consecutive red nodes due to property 3.

In other words, the longest path can never be more than twice the length of the shortest path.



The height of an RBT

- We want to show that the height h of a tree with n internal nodes is $O(\log_2 n)$.
- We specify internal nodes because the null leaves are not really nodes. The internal nodes are the keys.
- We will provide a sketch of the proof.
- The first step is the merge the tree – merge red nodes into the black ones.

Merging red nodes into black parents

- Consider this tree…

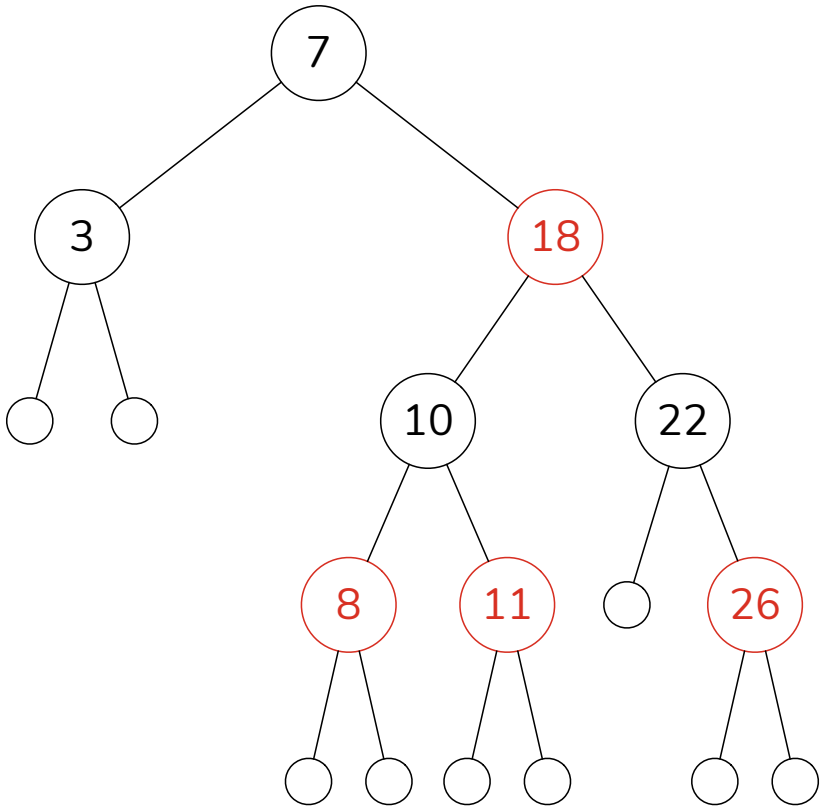
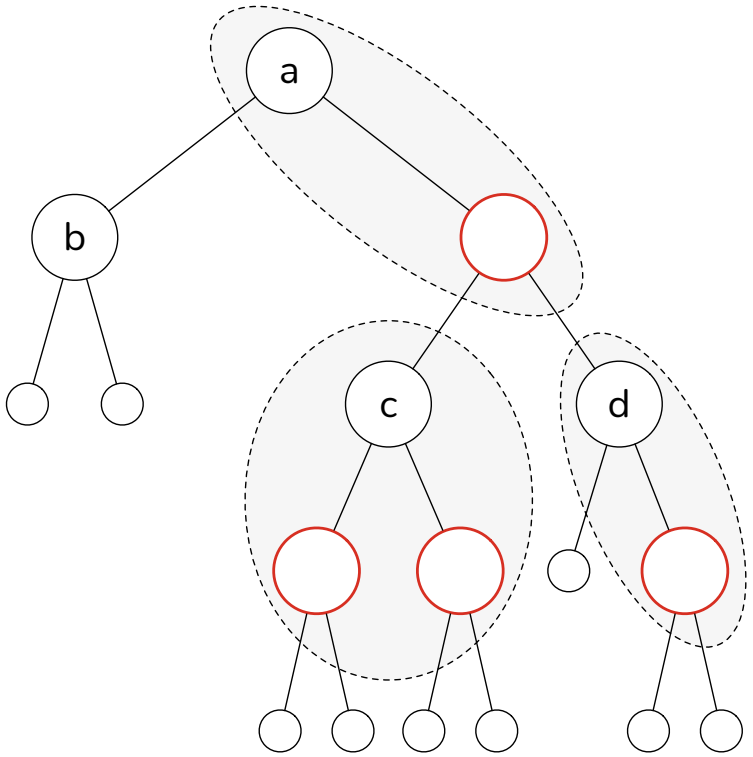
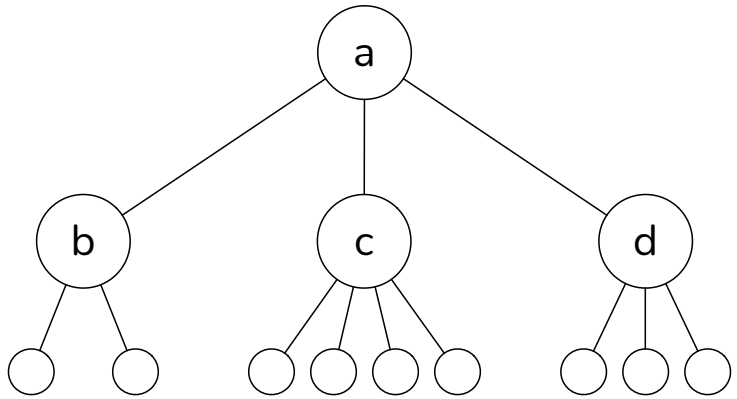


Image adapted from Eric Demaine

...continued

- After merging we get...





This is a 2-3-4 tree!

Image adapted from Eric Demaine

...continued

- After merging we get a 2-3-4 tree:
  - All nodes have 2, 3, or 4 children (except leaves who have none).
  - All leaves have the same height – because of property 4. This makes the tree balanced.
- Let:
  - h be the height of the original RBT.
  - h' be the height of the merged 2-3-4 tree.

The height of the merged tree

- Note that, in general, there are always $n + 1$ leaves in a tree that contains n internal (key) nodes.
- This is because every internal node has exactly 2 children.
- In the previous example, there are 8 internal nodes and 9 (i.e., $n + 1$) null leaves.

Proof: #Leaves = #Internal + 1

- Let n = number of internal nodes, and l = number of leaf nodes.
- Step 1: place 2 ‘tokens’ on each internal node…

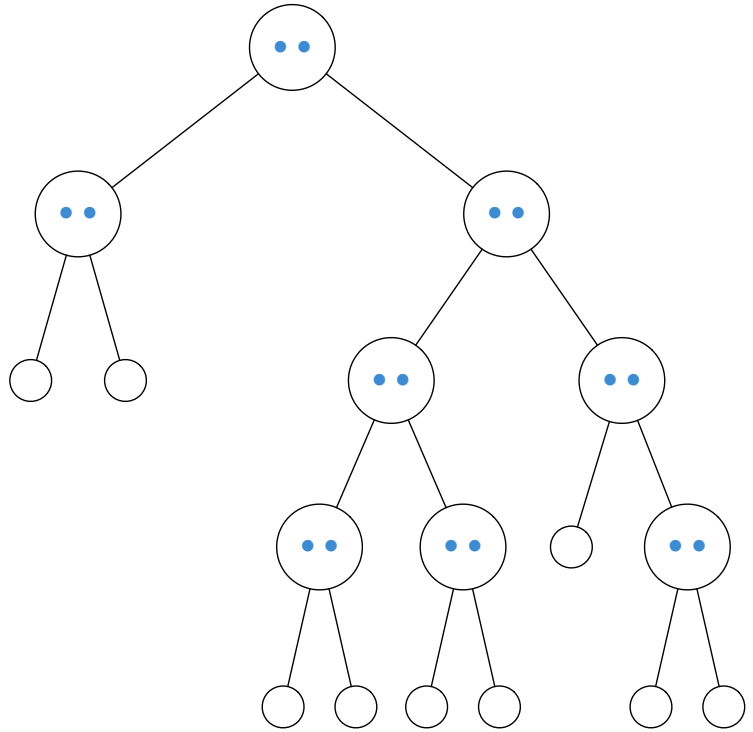


Image adapted from Eric Demaine

...continued

- Step 2: move one token to each child. Now all nodes except for the parent will have a token.

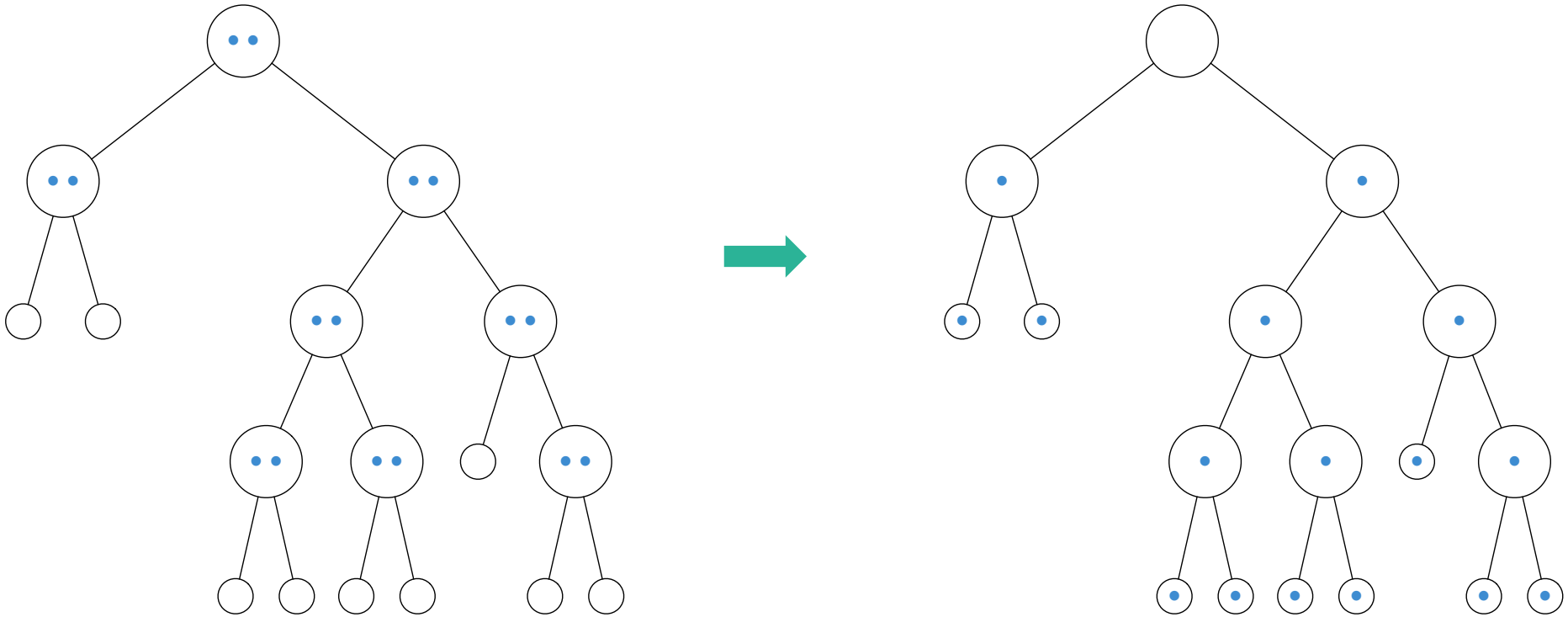


Image adapted from Eric Demaine

...continued

- We started with $2n$ tokens.
- We finished with l tokens (1 token per leaf) and $n-1$ tokens (1 token per internal excluding parent).
- Since we did not add or remove any tokens in the process, the number of tokens that we started with must be the same as when we finished...
- Rearrange:
  - $l + n - 1 = 2n$
  - $l + n - 1 = n + n$
  - $l - 1 = n$
  - $l = n + 1$

Height of RBT

- For a general 2-3-4 tree having height h':
  - The minimum number of leaves is $2^{h'}$ and the maximum number of leaves is $4^{h'}$.
- We know the merged red-black tree has $n+1$ leaves. So, the 2-3-4 tree has $n+1$ leaves too.
- So, we know that in my merged red-black tree, $2^{h'} \leq n+1$.
- If we log both sides, we get $h' \leq \log_2(n+1)$.
- Remember that the height of the merged RBT represents shortest possible path length (by rule 4) and earlier we showed that the longest possible path length is twice the shortest one.
- So, for the red-black tree having height h we get $h \leq 2 \times \log_2(n+1)$

So far...

- We have shown that we can perform operations in $O(\log_2 n)$ time if we follow the red-black tree rules.
- The next step is to create the insertion and deletion operations such that the red-black tree properties are not violated.

An insert strategy (this will not work in all cases)

- Use a normal BST search to find the insert location.
- We have a choice to make the node we insert either red or black.
- If we make the new node black, we violate rule 4 (the number of black nodes in any path is the same).
- If we make the node red, we might violate property 3 (consecutive reds).
- Since we must make a choice, we will go for red since a violation doesn't always occur, and such a violation happens to be easier to fix than a black violation.

A note on inserting

- If we insert 2 in this tree, we are OK. No rules are violated, and we are done (remember that we choose to always insert as red).
- In other words, if the parent is black, insertion is trivial.

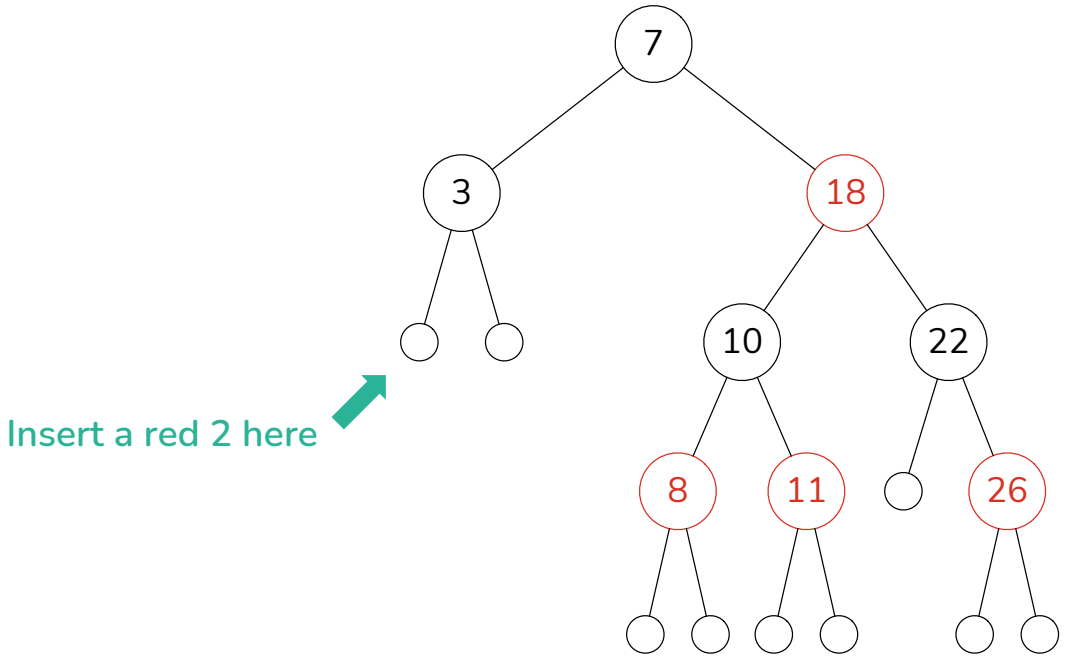


Image adapted from Eric Demaine

Inserting problem

- If we add 15, we have a problem (we get a 11-15 red+red violation).

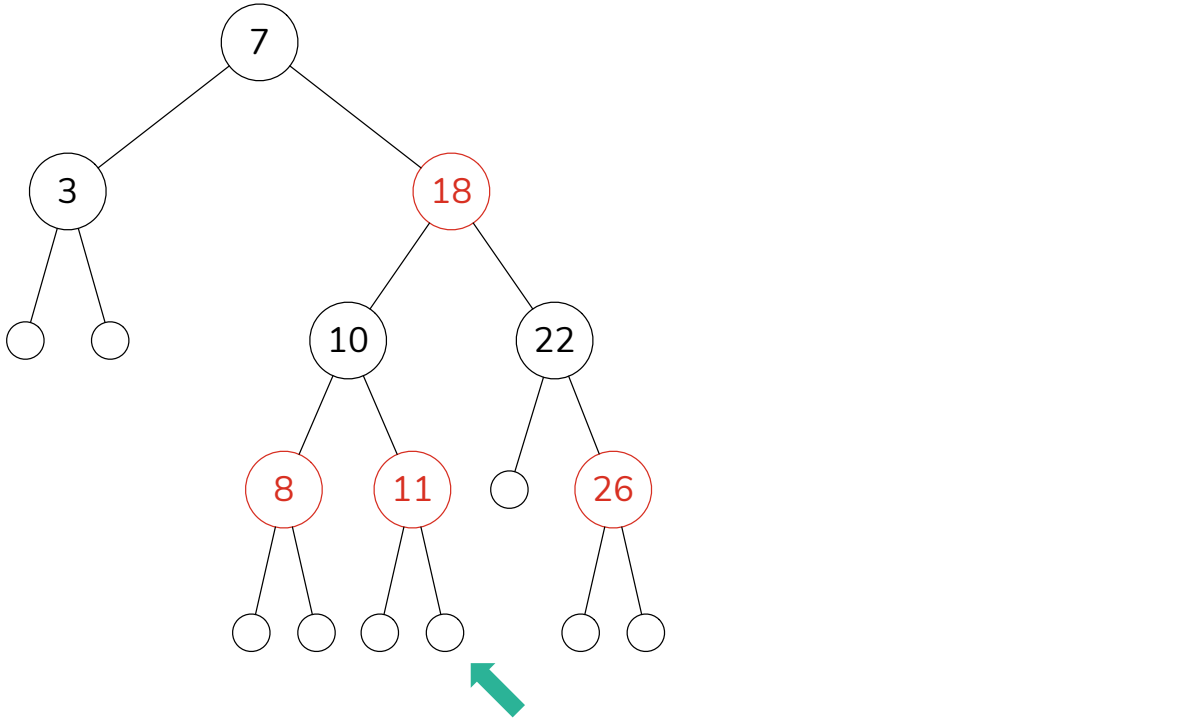


Image adapted from Eric Demaine

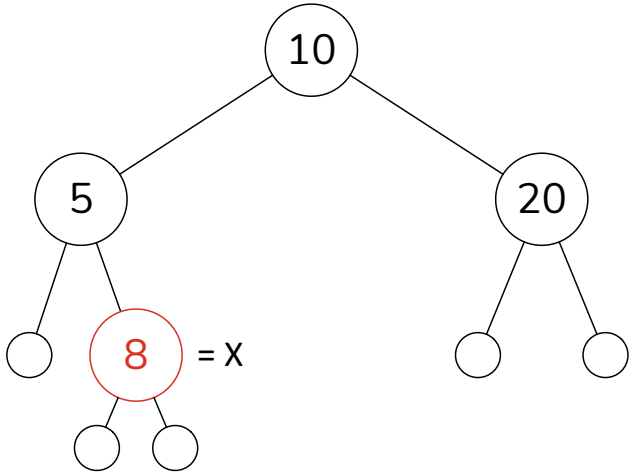
Our insert strategy so far…

- Always insert a red node X.
- If this is the root, colour it black and we are ready.
- If the parent P is black, we are ready.
- If the parent is red and the uncle U is black:
  - If relative to the grand parent G, X is outside:
    - Do an LL (or RR by symmetry if we inserted to right of P) and recolour old root (G) as red and new root to black.
  - If relative to G, X is inside:
    - Do an LR (or RL by symmetry) and recolour as above.
- If both the parent and uncle are red, then the scheme above will not work, and we will have to recursively fix further violations.

Some examples

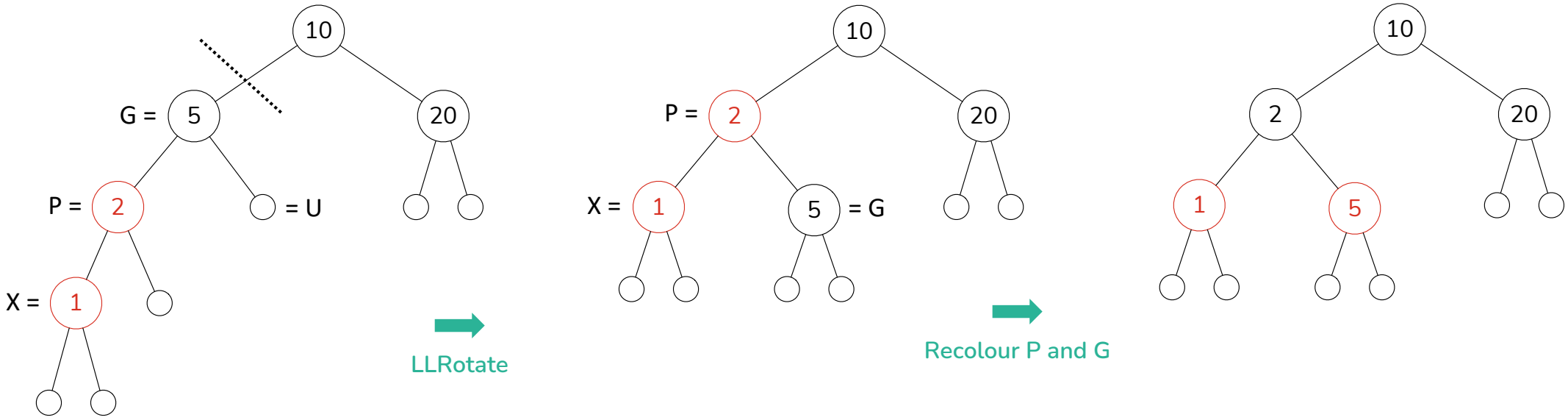
- The new node is the root – insert red X, recolour to black:

- The parent is black – insert red X and we're done:



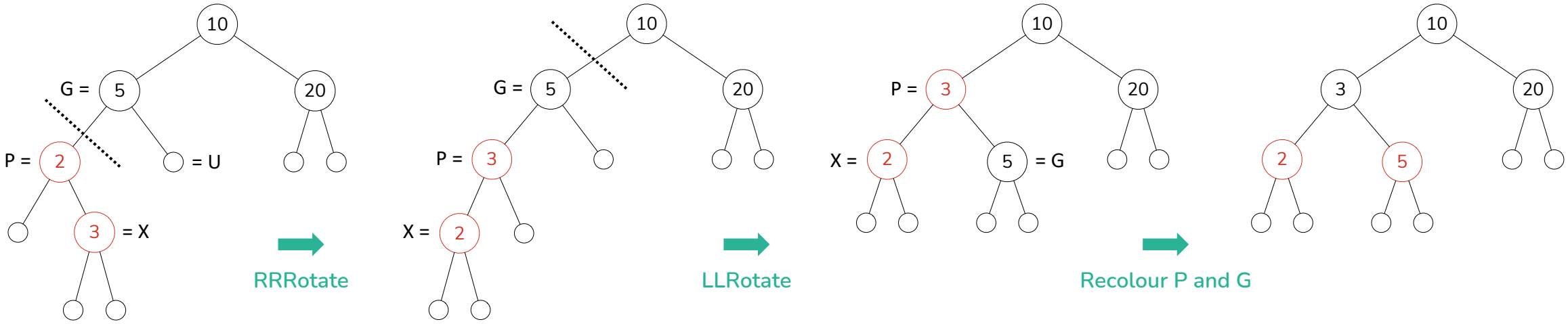
...continued

- Parent P is red, uncle U is black, X is outside grandparent G:



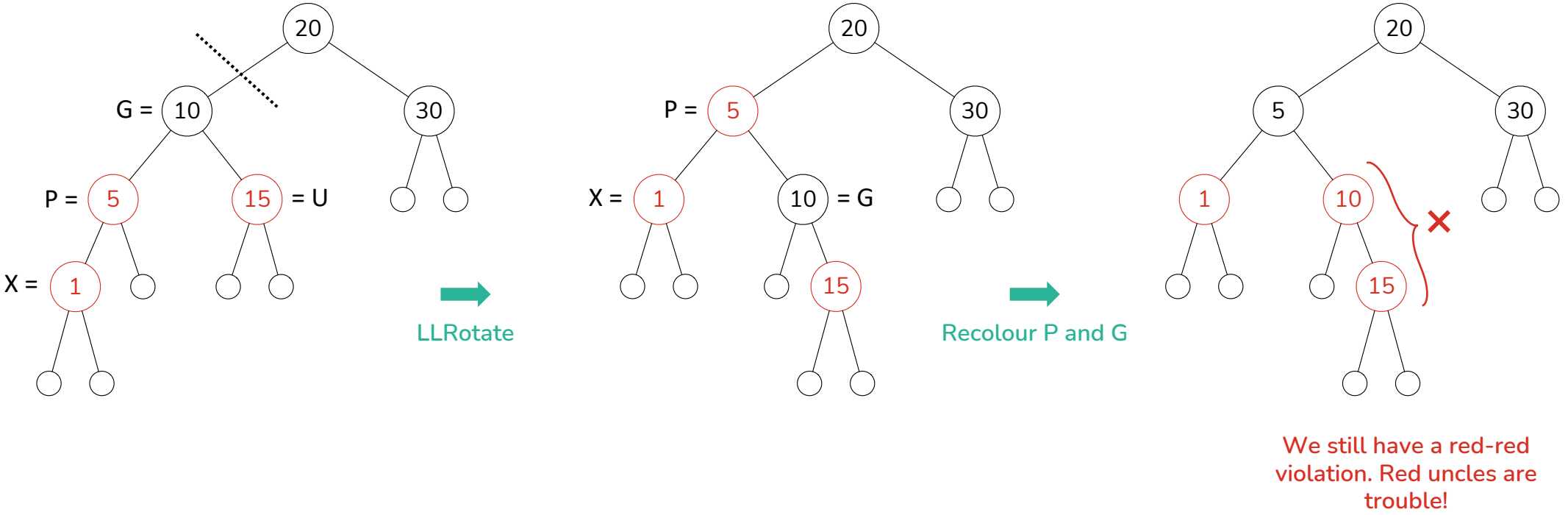
...continued

- Parent P is red, uncle U is black, X is inside G:



...continued

- Parent P is red, uncle U is red, X is outside G:



A working top-down insert

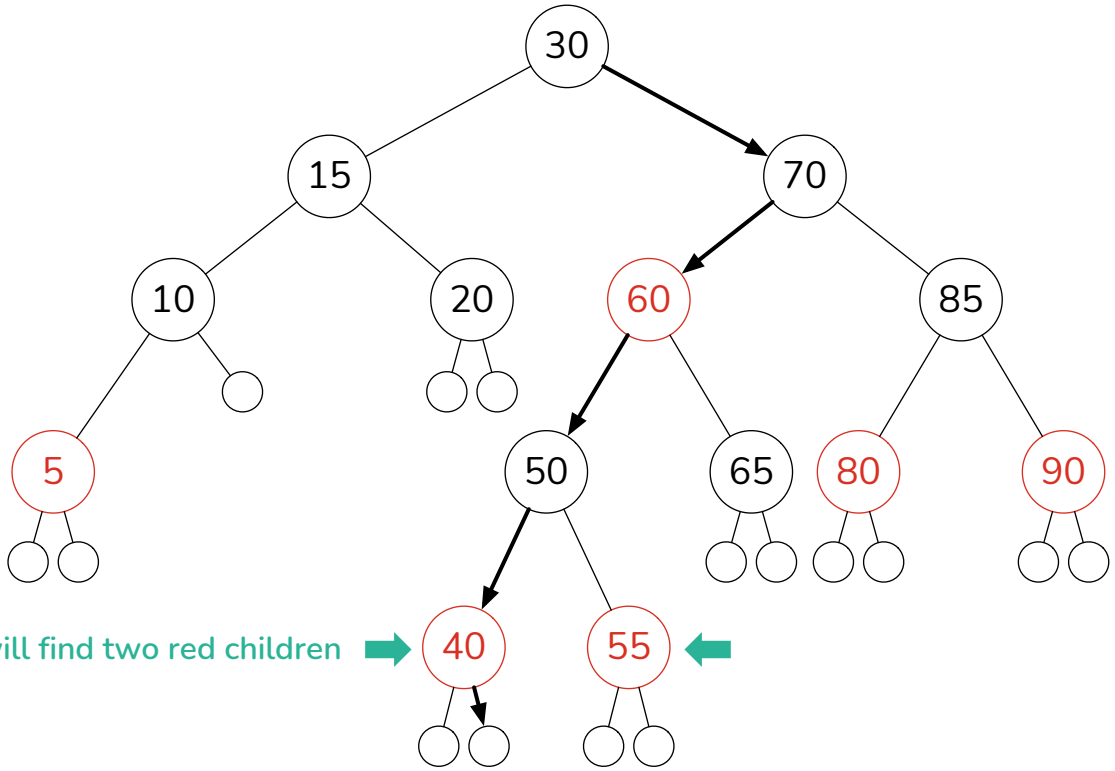
- When comparing AVL trees to RBTs, AVL trees are shown to be shallower.
  - AVL tree height <= 1.44 × $\log_2(n)$.
  - RBT height <= 2 × $\log_2(n)$.
  - So, for searches AVL trees may be slightly faster.
- AVL trees require a pass down to find the location of the new item and another pass up to rebalance the tree.
- So far, it appears that the RBT has the same problem: an insert and rebalance path is required.
- Additionally, RBTs so far require recursion to find the location and rebalance up.
- However, a modification solves this – top-down insertion.

...continued

- The basis of the algorithm is ensuring that an uncle is never red and then fix with one single or double rotation.
- When searching for the location where to place our new node X, if we find a node Y with 2 red children, we set Y to red and the children to black.
- If Y was the root, we will violate rule 2 but this is easily fixed at the end.
- This way we are guaranteeing that a red uncle can never exist.
- However, if Y's parent is red, we introduce a red-red violation. To fix this we do a single rotation (if outside) or a double rotation (if inside).

Example

- Let's start from this tree. We want to insert 45 and, on the way down, we see 50 which has two red children. Flip their colours.

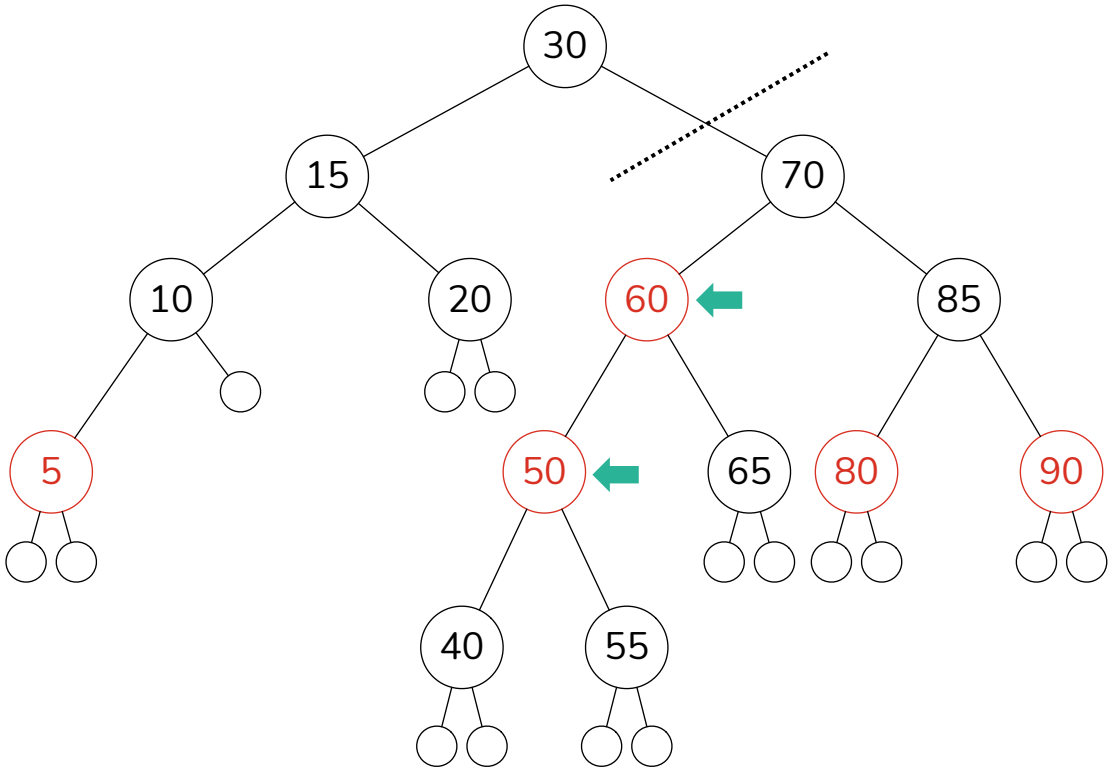


On the way down, at node 50, we will find two red children

Example adapted from Weiss

...continued

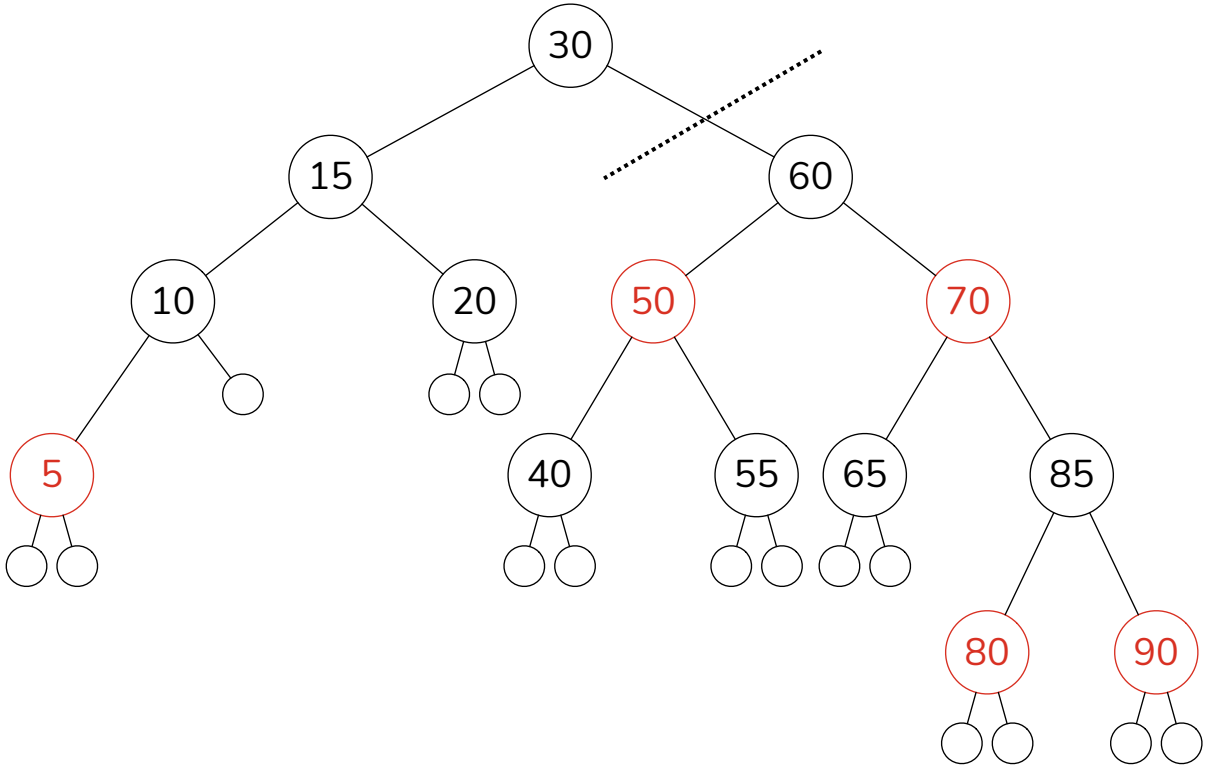
- Flipped children of 50 to black and set 50 to red. Now we have a violation at 60 and 50 because they are both red. 50 is outside of 70 so we do an LL rotation and recolour.



Example adapted from Weiss

...continued

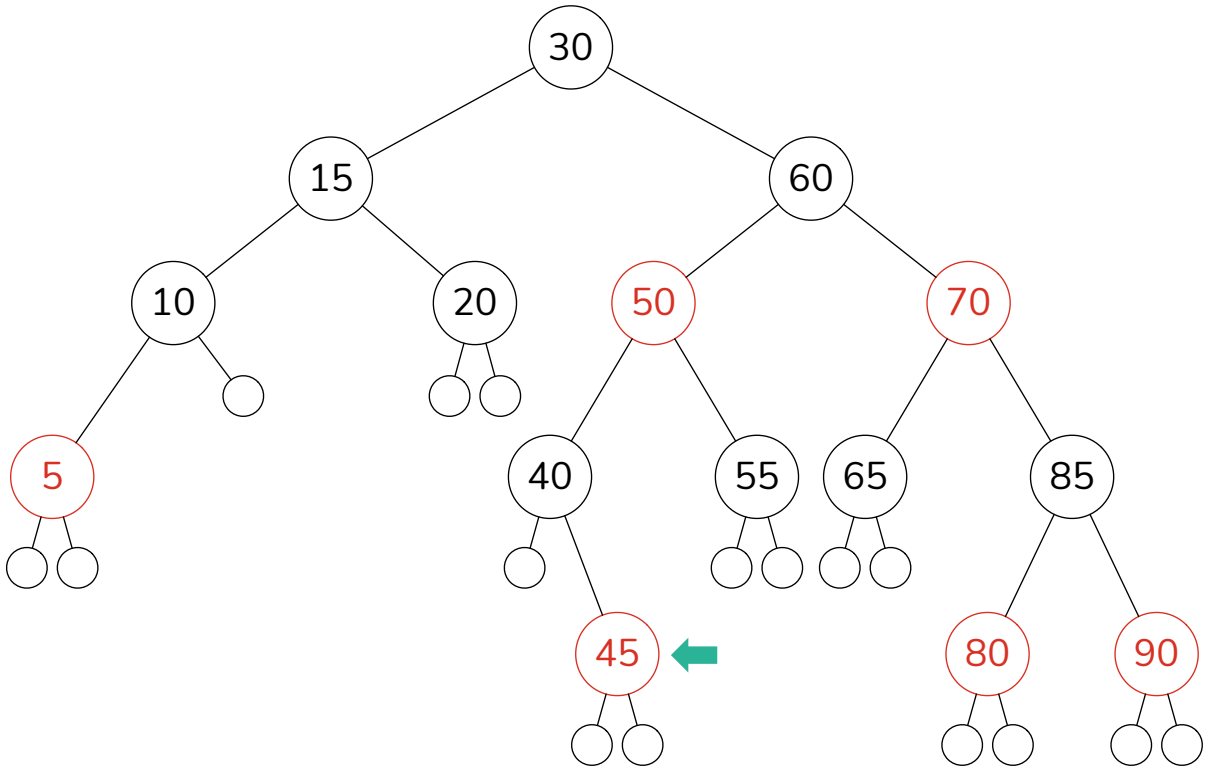
- Result of LL rotation and recolour at 70 fixes the problem.



Example adapted from Weiss

...continued

- We can proceed to insert 45. The parent is black, so we are done. If parent was red do another rotation. If the root was made red, make it black.



Example adapted from Weiss

Deletion

- Remember the 3 cases of ‘normal’ BST deletion.
- In RBT if we delete a red leaf node then there is no problem – it cannot violate any rules.
- If we delete a black node, we will cause a problem.
- We can implement a top-down deletion which is iterative and ‘fixes’ on the way down.
- Deletion is left as an exercise to the student (it is simple enough).

Further reading

- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
  - Red-Black Tree Online Videos (Erik Demaine, Massachusetts Institute of Technology, MIT, http://videolectures.net/mit6046jf05_demaine_lec10/)