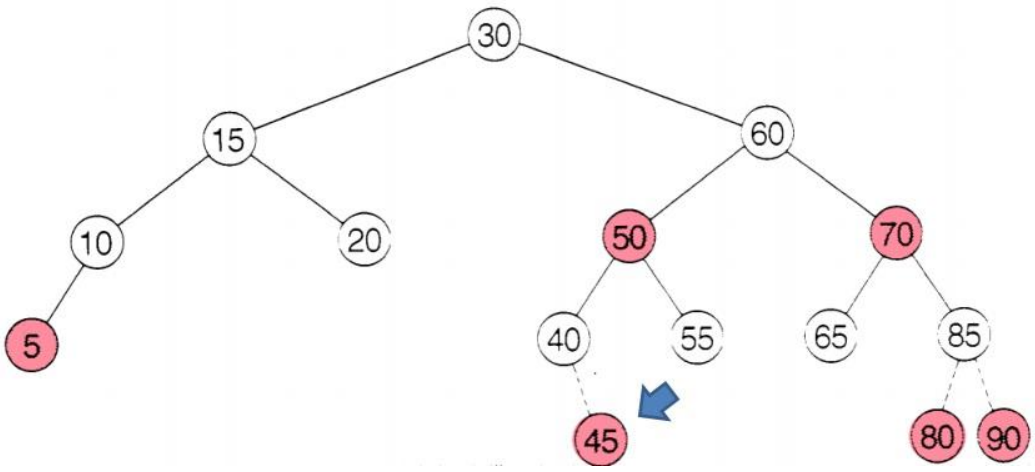
        - An RR rotation would be the same except with the “Left” and “Right” instances swapped.
        - Note that an LL or RR SINGLE rotation would not work for case 2 and case 3. A double rotation is required for these cases.
- Red-Black trees:
    - Red-Black trees are an improvement to AVL trees.
        - With AVL trees, an insertion requires the algorithm to recursively go down the tree to find the insertion point, and then go back up to retrieve height information.
        - Red-Black trees instead need just a single top-down pass, thanks to the red or black attribute that each node can have.
    - Properties:
        - Every node is “coloured” red or black.
        - Every leaf node is a NIL node (null) and can only be black.
        - If a node is red, then both its children are black.
        - Every simple path from the node to a descendant leaf contains the same number of black nodes.
        - The root node must be black.
    - If the above properties are followed, then we can perform operations in O(Lg(n)) time complexity.
    - Observations:
        - The shortest path will consist of only black nodes.
        - The longest path will consist of alternating red and black nodes.
        - The longest path can never be more than twice the length of the shortest path.
- Red-Black tree insertion:
    - A new node must always start as red.
    - If the new node is the root, then colour it black.
    - If the parent is black, then the insertion was successful.
    - Otherwise, if the parent is red and the UNCLE is black:
        - If, relative to grandparent G, the node is on the outside, then perform an LL or RR rotation (depending on whether the node inserted is to the left or right of the node’s parent), and recolour the old root G as red and the new root as black.
        - If, relative to grandparent G, the node is on the inside, then perform an LR (LL + RR) or RL (RR + LL) rotation (depending on whether the node inserted is to the left or right of the node’s parent), and recolour as above.
    - If both the parent and uncle are red, then the scheme above will not necessarily work, and we will have to recursively fix further violations through additional recolouring.

- Finally, the node with value 45 is added:



- If more red violations are caused, then the rotation is performed again. The root is also checked to ensure it is black. In this case, we are done.

- Red-Black tree deletion:
  - There are multiple deletion cases for red-black trees.
  - The easiest case is if the node to be deleted is red. In this case, it can simply be deleted as if it were a normal node of a BST, and it wouldn't violate any rules.
  - Deleting black nodes is where the problems begin. Top-down deletion iteratively starts at the top and works its way down, rebalancing and recolouring the tree.
  - Conventions: let X be the node being deleted, T be X's sibling, P be X's parent, R be T's right child and L be T's left child. Let us also always assume that X is P's left child. The same rules apply except reversed for right childs.
- RBT deletion process:
  - Step 1:
    - Start from the root.
    - Make the root red.
    - If both children are black:
      - Move X to the appropriate child via the BST search.
      - Go to Step 2.
    - Otherwise, if one or more of the root's children are red:
      - Mark the root as X.
      - Go to Step 2B.
  - Step 2:
    - If X has 2 black children:
      - Go to Step 2A.
    - If X has at least one red child:
      - Go to Step 2B.

- One set of data can have multiple list "levels".
- The higher the level is, the less elements there are (i.e. the more it "skips"). The lowest level is the full list that contains all of the data elements.
- How do we decide which elements are in the higher list levels?
  - If this were a subway, then the higher lists ("express" routes) would contain the most popular stops.
  - However, we don't have this information for our skip lists, so the ideal option is to spread out the skips uniformly.

- Skip list searching:
  - Initialisation: Let L = topmost list.
  - Traverse the list from L until we find the item.
  - If we reach an item larger than the one we're looking for (i.e. "we got too far"), then:
    - If this is the bottom of the list, then the item isn't found.
    - Else, move to a lower list.
  - Restart from the second step.
- Optimising the lists:
  - With a uniform spread and two lists (L1 and L2, where L1 is the higher-level list), the worst case scenario, which is the last item, would require:
    - Going through |L1| stops.
    - Going down to |L2| and going through |L2| / |L1| stops.
  - Therefore, the amount of steps taken would be |L1| + |L2| / |L1|.
  - We want to minimise this value to optimise the searching. This is done by choosing the right size of L1. The formula will be optimised when:

|L1| + |L2| / |L1|

This is smallest.

This is largest.

  - This is optimised when |L1| = √|L2|
  - For example: if we have 150 items in L2, then if we have √150 elements in L1, we'd have √150 + 150/ √150 = 24.5 data items to travel through in the worst case scenario.
  - This gives us a time complexity of Θ(√n) (worst case). This is better than Θ(n) which is a linked list's time complexity, but it can be improved by adding more "express lines", i.e. higher list levels.
  - The optimised amount of lists needed would be lg(n), which gives logarithmic time complexity if you work it out.
- Skip list insertion/deletion:
  - What is written above is all about initialising and searching through the skip list in an ideal fashion. However, insertions and deletions can tamper with the skip list, essentially "corrupting" it.