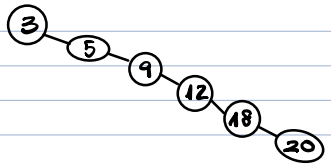
BALANCED BINARY SEARCH TREES

(AVL & Red/Black trees)

BALANCING BSTs

- As operations are performed on a BST it could become highly unbalanced (a degenerate tree)



- Our implementation does not ensure the BST stays balanced.
- Other approaches do, such as AVL trees, red/black trees and 2-4 trees.
- We'll explore rotations - operations on BSTs to assist in the process of keeping a tree balanced.

ORDERED LIST ANALYSIS ⇒

Operation	LinkedList	BinarySearchTreeList
`removeFirst`	O(1)	O(log n)
`removeLast`	O(n)	O(log n)
`remove`	O(n)	O(log n)*
`first`	O(1)	O(log n)
`last`	O(n)	O(log n)
`contains`	O(n)	O(log n)
`isEmpty`	O(1)	O(1)
`size`	O(1)	O(1)
`add`	O(n)	O(log n)*

*both the `add` and `remove` operations may cause the tree to become unbalanced

SELF BALANCING TREES:

○ AVL TREES

- AVL tree ensures a BST stays balanced
- For each node in the tree, there is a numeric balance factor - the difference between the heights of its subtrees
- After each add or removal, the balance factors are checked, and rotations performed as needed.

- Insert/erase: update balance of each subtree from point of change to the root.

- The height of a tree is the number of nodes in the longest path from the root to a leaf node.

  ↳ Height of empty tree is 0: `ht(empty) = 0`

  ↳ Height of others: `ht(n) = 1 + max(ht(n.left), ht(n.right))`

- `Balance(n) = ht(n.right) - ht(n.left)` → restrict balance to -1, 0, +1.

Rotation brings unbalanced tree back into balance

AVL Tree Insertion

- We consider cases where new node is inserted into the left subtree of a node n
  - ↳ Insertion into right subtree is symmetrical.

- Case 1: the left subtree does not increase.
  - → No action necessary at n.

- Case 2: subtree height increases, and balance(n) = +1 or 0
  - → Decrement balance(n) to 0 or -1.

- Case 3: subtree height increases, and balance(n) = -1
  - → Need more work to obtain balance (because balance is now -2).

AVL Tree Insertion Rebalancing

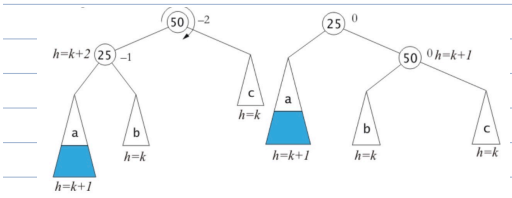
Cases:

- Case 3a: Left subtree of left child grew: Left-Left heavy Tree.
- Case 3b: Right subtree of left child grew: Left-right heavy tree

Rebalancing a Left-Left tree

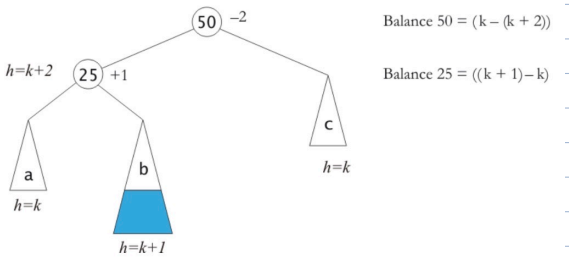
* Actual heights of subtrees are unimportant
  - ↳ Only difference in height matters when balancing

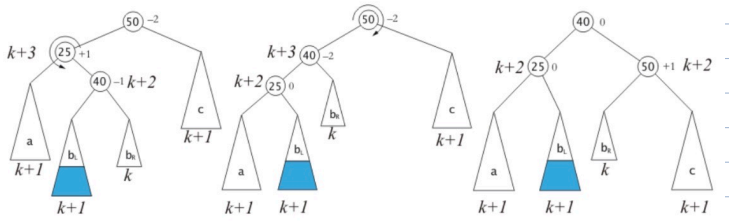
* In Left-Left tree, root and subtree are left-heavy
* One right rotation regains balance



Rebalancing a Left-Right tree

* Root is left heavy, left subtree is right-heavy
* A simple right rotation cannot fix this.
* Need:
  - Left rotation around child, then
  - Right rotation around root.





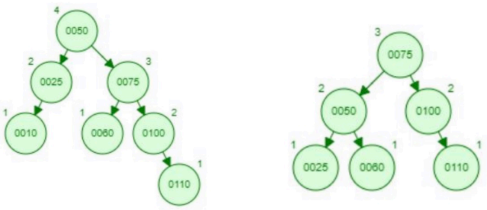
4 Critically Unbalanced Trees

- Left-Left (parent balance is -2, left child balance is -1) → Rotate right around parent
- Left-Right (parent balance -2, left child balance +1) → Rotate left around child
  ↳ Rotate right around parent
- Right-Right (parent balance +2, right child balance +1) → Rotate left around parent
- Right-Left (parent balance +2, right child balance -1) → Rotate right around child
  ↳ Rotate left around parent

Removing in an AVL Tree

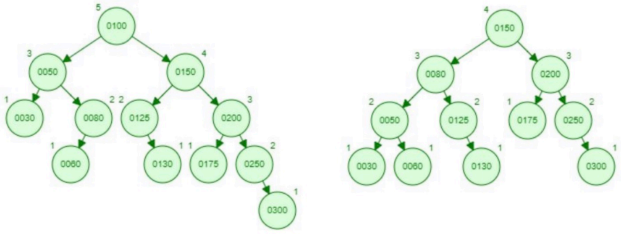
- Leaf and single parent: check from removing point and up.
- No balance is changed below the inserting point but could be changed upwards (at least the parent of the deleted node will always change balance).
- Follow the ancestors:
  * If the balance does not change: stop
  * If the balance changes:
    → and is -1, 0 or 1: continue to parent (if root stop).
    → and is -2 or 2: rotate and stop.

Remove a leaf or single parent is simple: check up from the point of deletion.



Check the balance of 25 (will change from -1 to zero)
Since it was changed check the balance of 50 (will change from 1 to 2)

Deleting an internal note: check from the replacement note and up.



Deleting 100: replacement node is 80. Check from 60 and up.
50 will change to zero. 80 will change to 2. Do a left rotation around 80 and stop.

Performance of AVL Trees

- Worst case height: 1.44 $\lceil \log n \rceil$
- Thus, lookup, insert, remove all: $O(\log n)$
- Empirical cost is $0.25 + \log n$ comparisons to insert

RED/BLACK TREES

- Another balanced BST approach. Better for dynamic trees.
- Each node has a color, usually implemented as a boolean value
- RULES:
  - * The root is black
  - * All children of red nodes are black
  - * Every path from the root to a leaf contains the same number of black nodes

Insertion

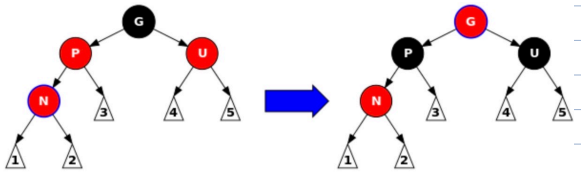
New nodes are coloured red and inserted as a standard BST.

4 scenarios:

1. N is the root node ; i.e., first node of red-black tree → Just change the color to black

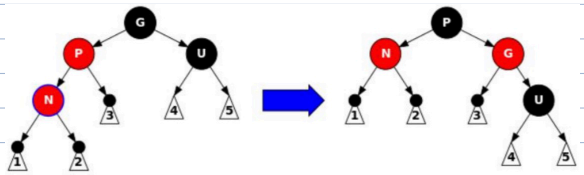
2. N's parent (P) is black → Everything is fine: no invariant is violated

3. P is red (so it can't be the root of the tree) and N's uncle (U) is red

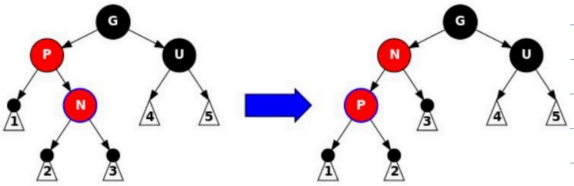


→ Change the color of the parent and uncle (possibly also grandparent if it's the root)

4. P is red and U is black.



→ Rotate around the grandparent and then change the color of parent and the grandparent.



→ If the tree is left-right heavy do a left rotation of the parent first (and mirror if it is right-left heavy).