ADT Dictionary

Dictionary: a **collection of items**, each of which **contains a key and a value**

↳ called a "**key-value pair**" (KVP)

↳ **Keys can be compared** and are (typically) unique.

Operations:

- **Search(k)**, aka, lookup(k)
- **insert(k, v)**
- **delete(k)**, aka, remove(k)

optionals: successor, merge, is-empty, size, etc

Common Assumptions:

1) Dictionary has **n KVPs**
2) Each KVP uses **constant space**
3) Keys can be compared in **constant time**
4) (Usually:) dictionary is **non-empty** before and after operation

	search	insert	delete
unsorted list/array	$O(n)$	$O(1)$	$O(1)$
sorted list/array	$O(\log n)$	$O(n)$	$O(n)$
binary search tree	$O(\text{height})$	$O(\text{height})$	$O(\text{height})$

Review: binary search

↳ note: only applies to a sorted array!

```
binary-search(A, n, k):
1) l=0, r=n-1
2) while (l < r) {
3)     m = ⌊(l+r)/2⌋
4)     if (A[m]==k) return "found at A[m]"
5)     else if (A[m] < k) then l=m+1
6)     else r=m-1
7) }
8) return "not found :c, but would be between A[l-1] and A[l]"
```

Review: Binary Search Trees (BSTs)

Structure:

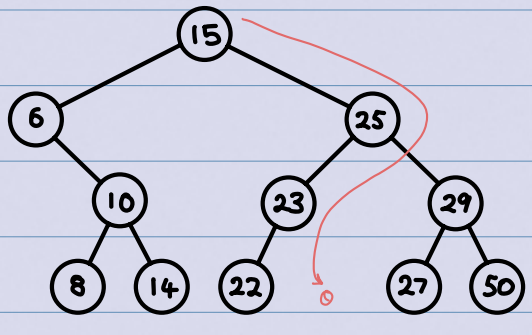
- all nodes have two (possibly empty) subtrees
- every node stores a KVP
- empty subtrees usually not shown

Ordering:

- every key k in T.left is less than the root key
- every key k in T.right is greater than the root key

BST::Search(k) → start at root, compare k to current node's key. Stop if found or subtree is empty, else recurse at subtree.

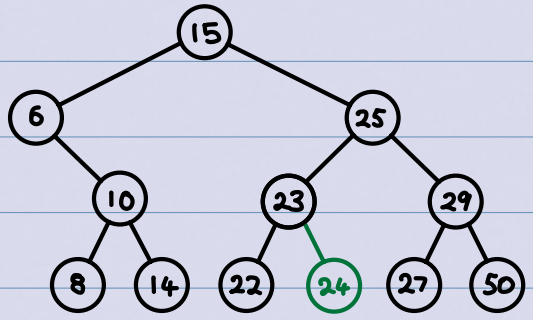
Eg: BST::Search(24):



∴, 24 not found in the BST!

BST:: insert(k, v) → Search for k, then insert (k, v) as a new node.

Eg: BST:: insert(24, v):



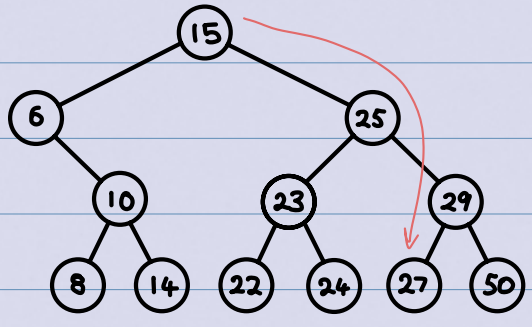
∴, 24 inserted!

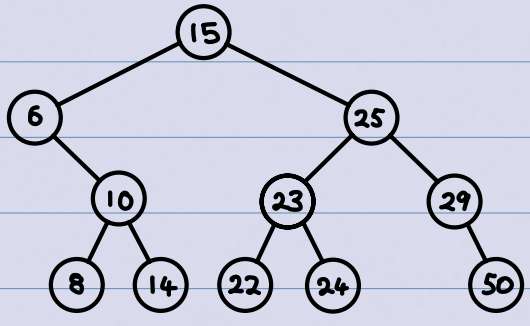
BST:: Delete(k) → first search for the node x that contains the key.

- If x is a leaf, delete it.
- If x has one empty subtree, move the child up
- Else, swap key at x with key at successor node and then delete that node

- Successor: next-smallest among all keys in the dict.

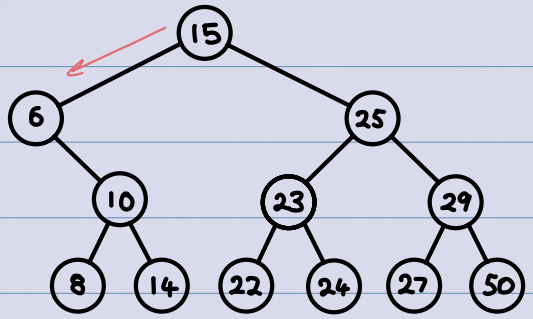
Eg: BST:: Delete(27): (leaf)

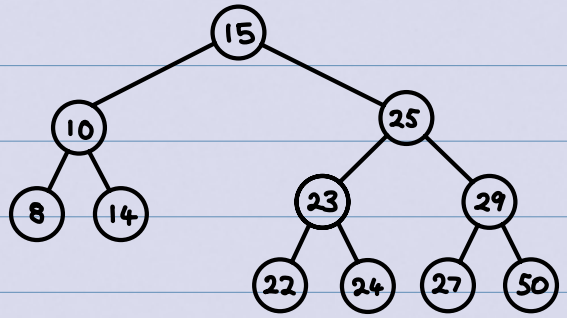




∴ Deleted using case 1

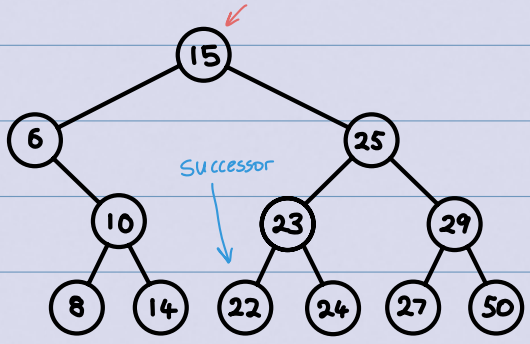
Eg: BST:: Delete(6):

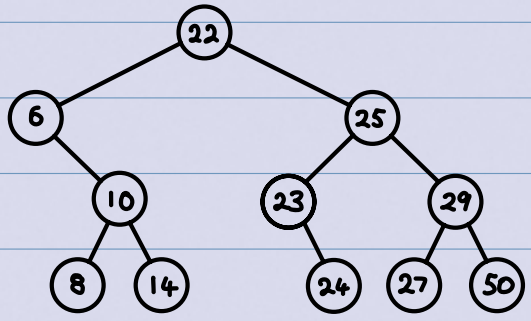




∴ Deleted using case 2

Eg: BST::Delete(15)





∴, deleted using case 3

→ BST::Search, BST::insert, and BST::delete all $O(h)$, where h = maximum number for which level h contains nodes.

↳ single-node tree has height 0, empty tree has height of -1.

- if n items are inserted one-at-a-time, how big is h?

↳ worst-case: $n-1 = O(n)$

↳ best-case: $O(\log n)$

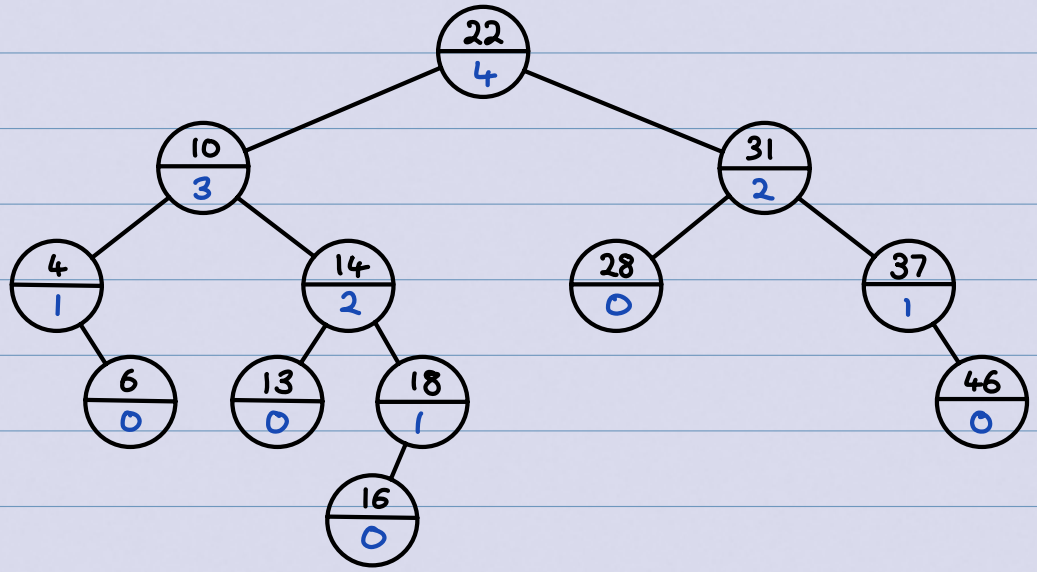
AVL Trees

an AVL tree is a BST with an additional height-balance property at every node:

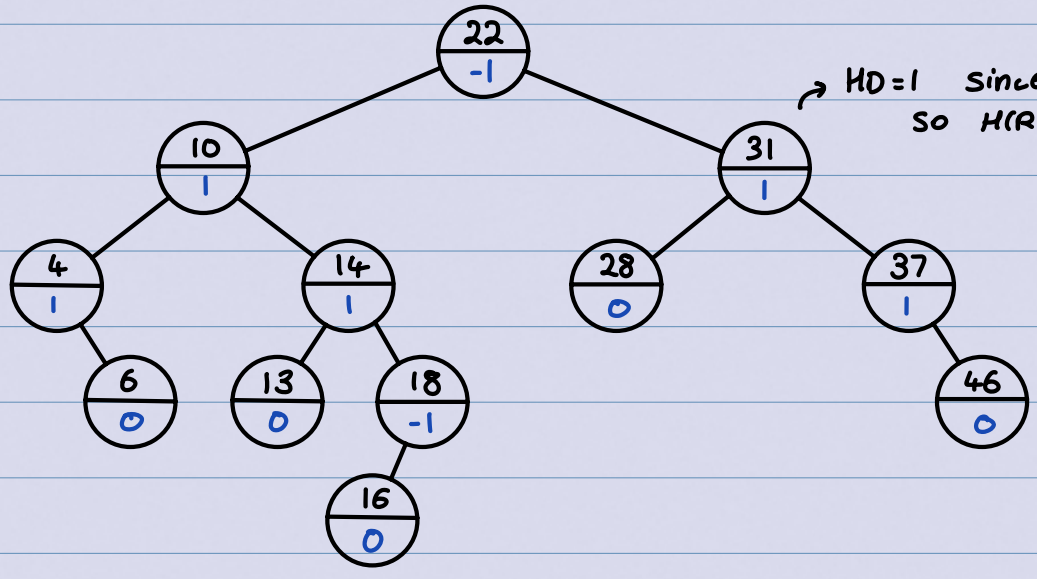
The heights of the left and right subtree differ by at most 1.

↳ ie, if node z has left subtree L and right subtree R, then height(R) - height(L) must be in {-1, 0, 1}

AVL Tree Example (w/ height):



AVL Tree Example (w/ height-difference):



Theorem: the height of an AVL tree on n nodes is in $\Theta(\log n)$.

↳ BST::search, BST::insert, BST::delete all cost $\Theta(\log n)$ in the worst case.

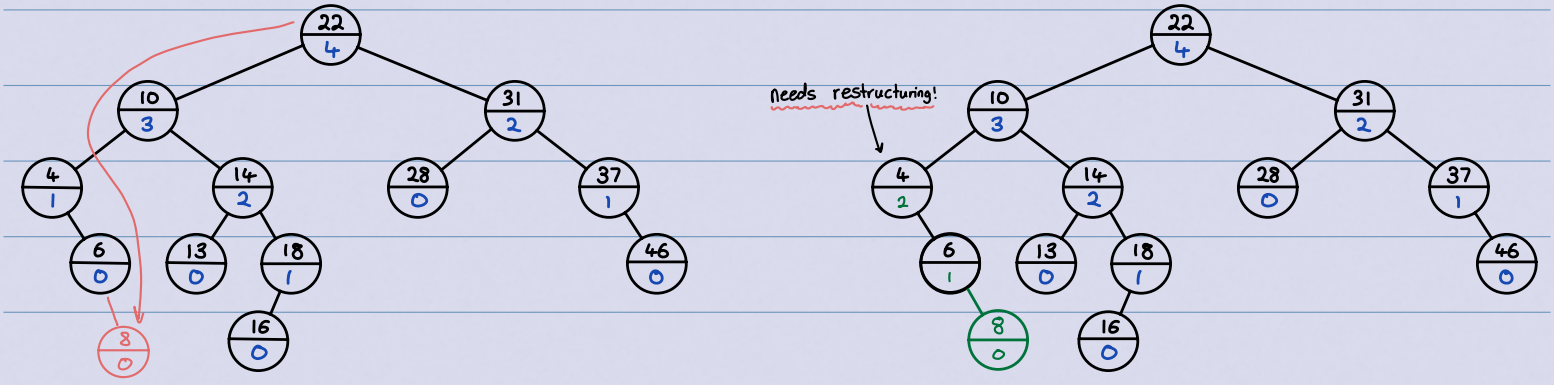
AVL::insert(k,v) → first, insert(k,v) with usual BST

insertion. We assume that this returns the new leaf z where the key was sorted. Then, move up the tree from z, and update the height (easy to do in constant time). If the height difference becomes ±2 at node z, then z is unbalanced, so we must re-structure the tree.

→ note, to set the height, we simply do:

u.height = 1 + max {u.left.height, u.right.height}

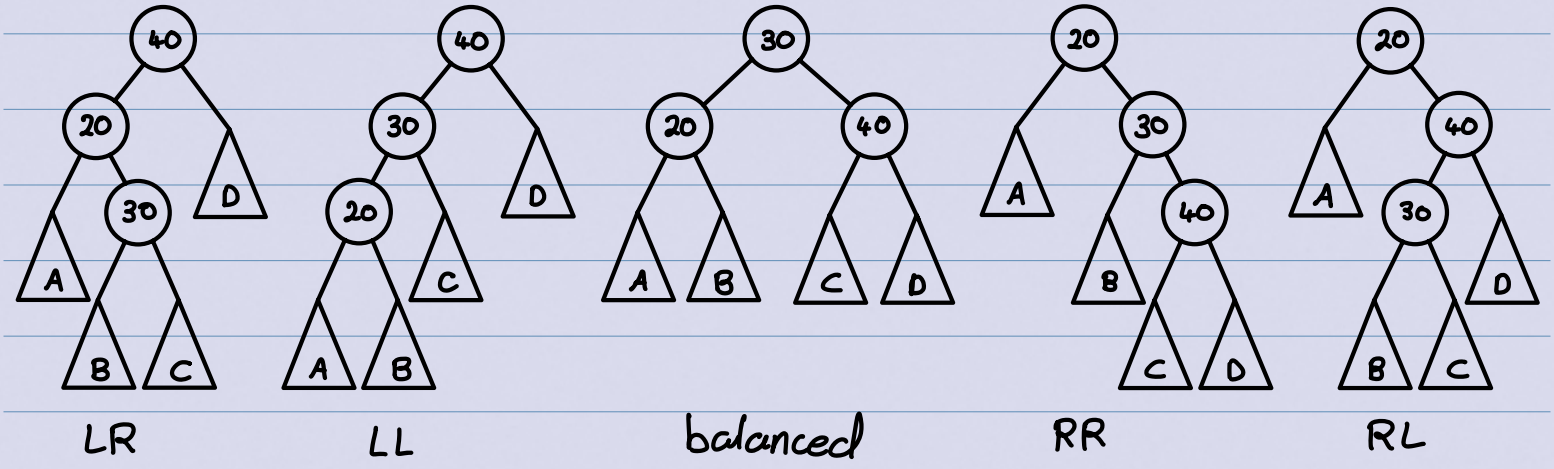
Eg: AVL: insert (8):



∴ After restructuring (later), 8 is inserted correctly!

Restructuring in a BST: Rotations

There are many different BSTs with the same keys. Eg:

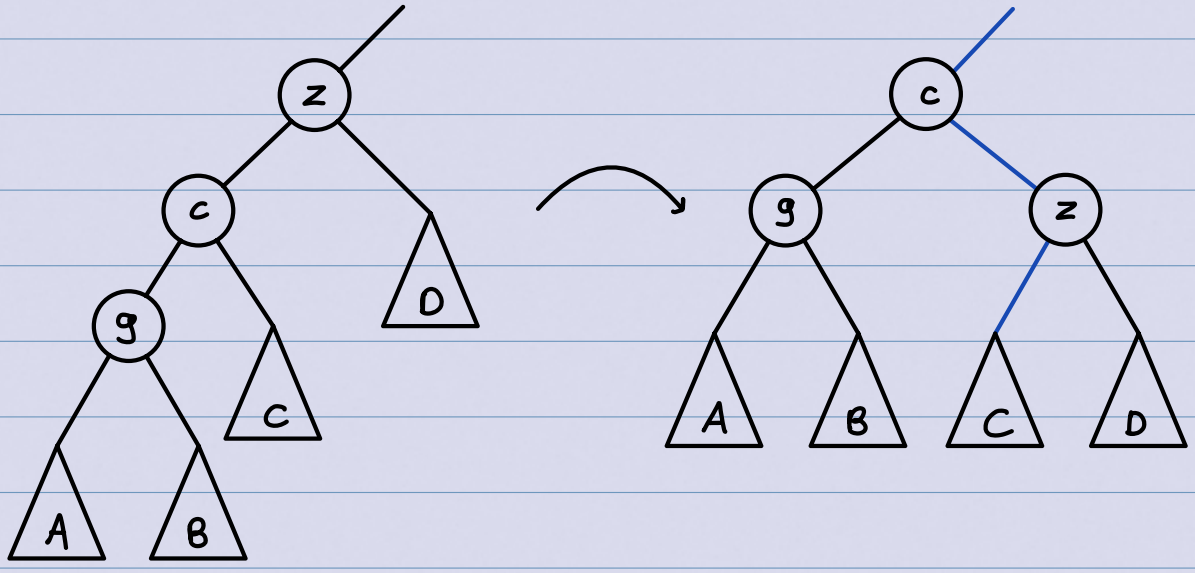


→ goal: change the structure without changing the order, and

restructure such that the subtree becomes balanced.

Right Rotation

this is a right-rotation on node z:



→ only O(1) links are changed. Useful to fix left-left imbalances.

Right-Rotation Pseudocode:

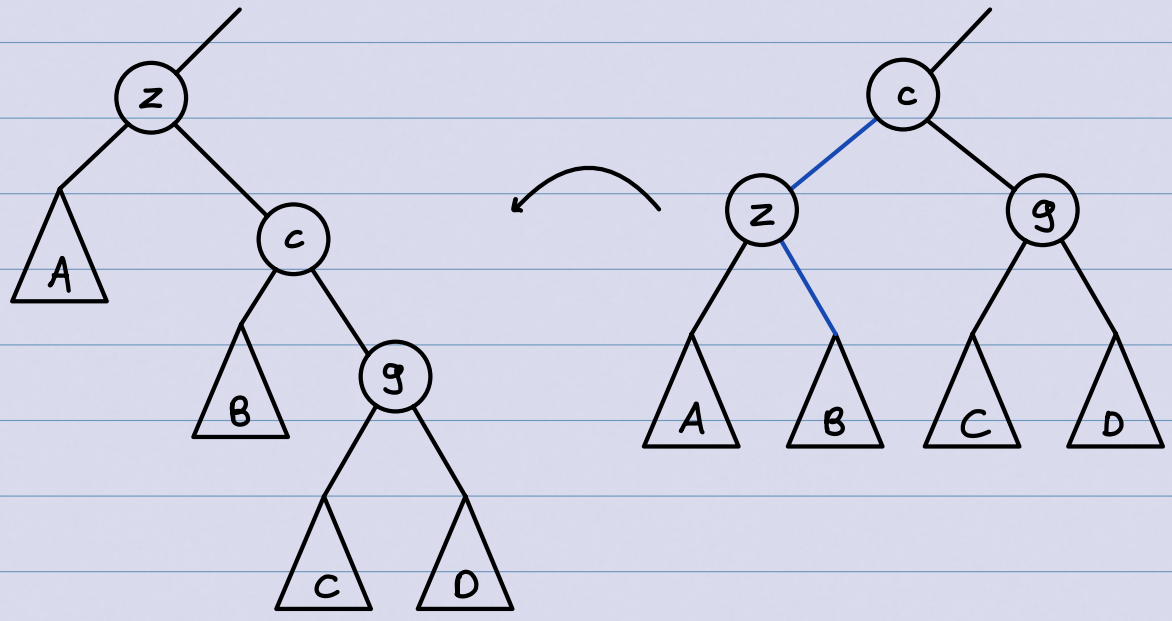
```
rotate-right(z)
1) c = z.left
2) // fix links connecting to above
3) c.parent = (p = z.parent)
4) if (p == null) { root = c }
5) else {
6)     if (p.left == z) { p.left = c }
7)     else { p.right = c }
8) }
9) // actual rotation
10) z.left = c.right,    c.right.parent = z
11) c.right = z,    z.parent = c
12) set-height-from-subtrees(z),    set-height-from-subtrees(c)
```

```
13) return c // returns new root of subtree
```

↳ runs in O(1)!

Left Rotation

this is a left-rotation on node z:



Again, only O(1) links need to be changed. Useful to fix right-right imbalances.

Left-Rotation Pseudocode:

```
rotate-left (z)
1) c = z.right
2) // fix links connecting to above
3) c.parent = (p = z.parent)
4) if (p == null) { root = c }
5) else {
6)     if (p.right == z) { p.right = c }
7)     else { p.left = c }
8) }
9) // actual rotation
```

10) z.right = c.left, c.left.parent = z

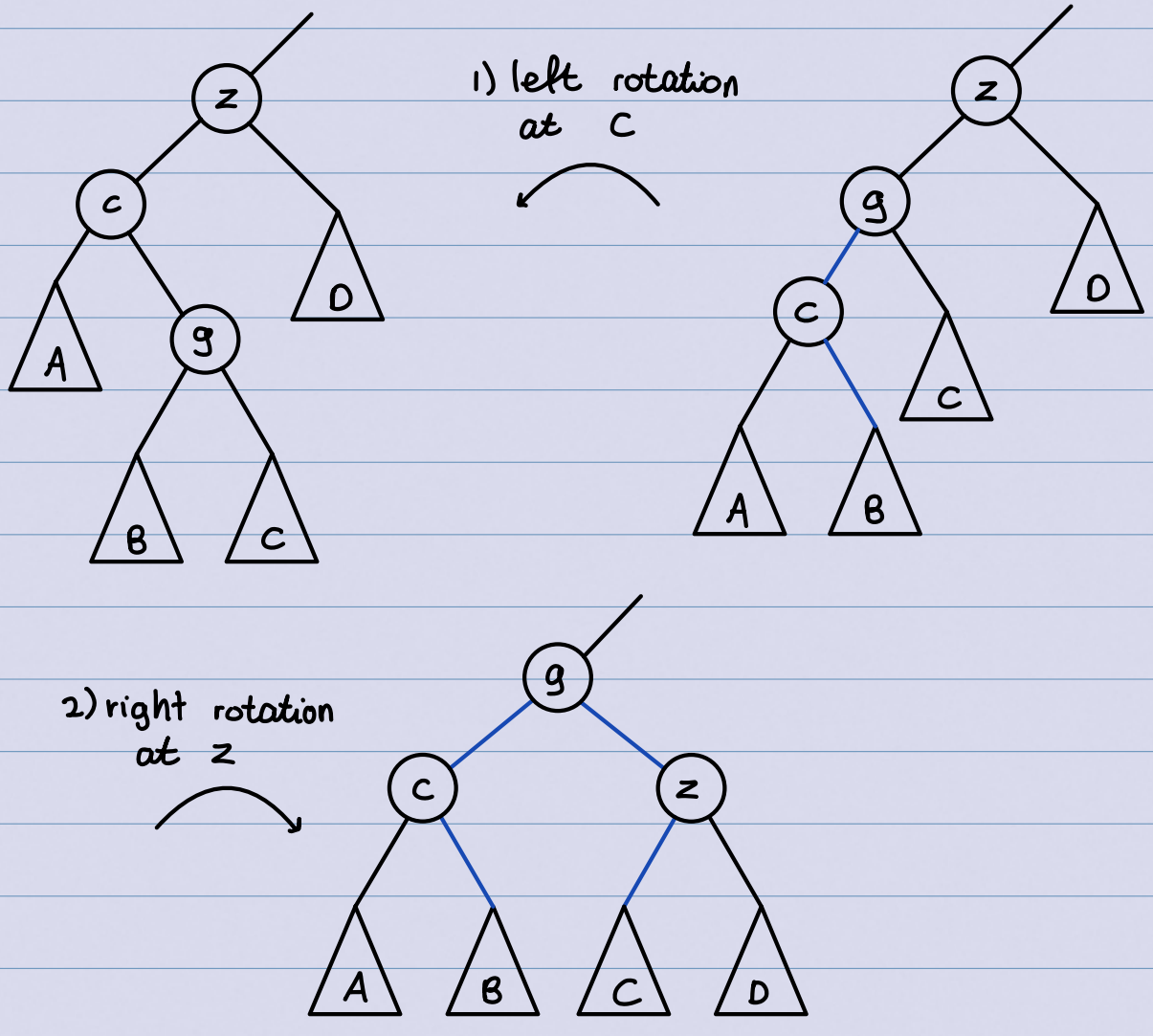
11) c.left = z, z.parent = c

12) set-height-from-subtrees(z), set-height-from-subtrees(c)

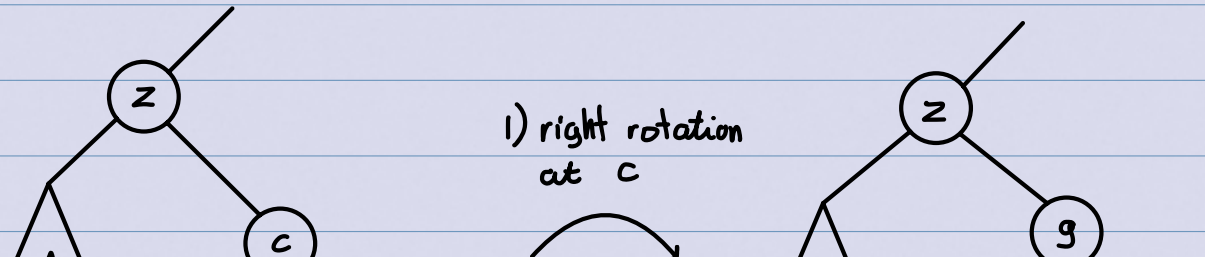
13) return c // returns new root of subtree

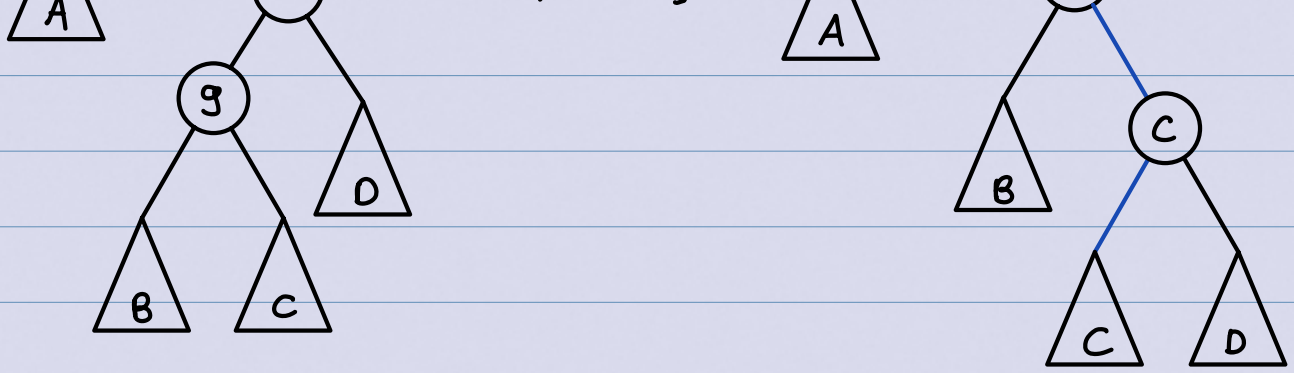
↳ runs in O(1)!

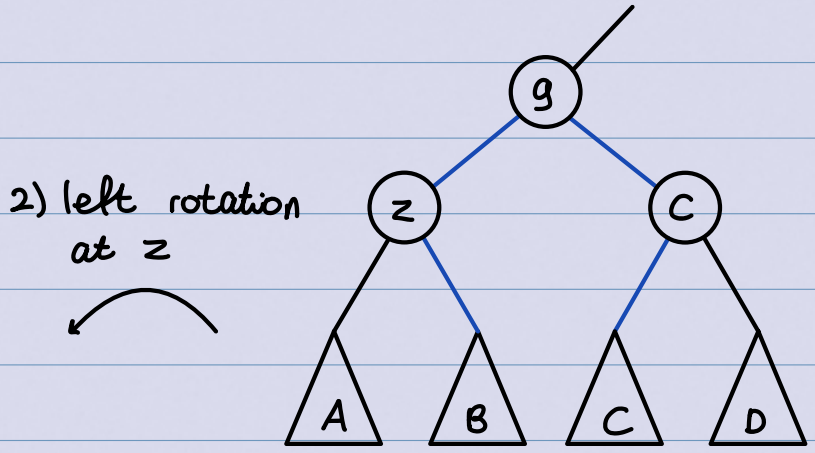
Double Right Rotation



Double Left Rotation







AVL Insertion Revisited

Imbalance at z: do (single or double) rotation

- Choose c as child where subtree has bigger height.

```
AVL::insert(k,v)
1)  z = BST::insert(k, v)      // new leaf with k
2)  while (z is not null) {
3)    if ( |z.left.height - z.right.height| > 1) {
4)       c = taller child of z
5)       g = taller child of c    //grandchild of z
6)       z = restructure(g, c, z)  // see in next code block
7)       break
8)    }
9)    set-height-from-subtrees (z)
10)   z = z.parent
11) }
```

↳ for insertion, one rotation restores all heights of subtrees.

Fixing an slightly-unbalanced AVL tree:

restructure(g, c, z) → node g is child of c which is child of z

1) case

// right rotation
u = rotate-right(z)

// double-right rotation
rotate-left(c)
u = rotate-right(z)

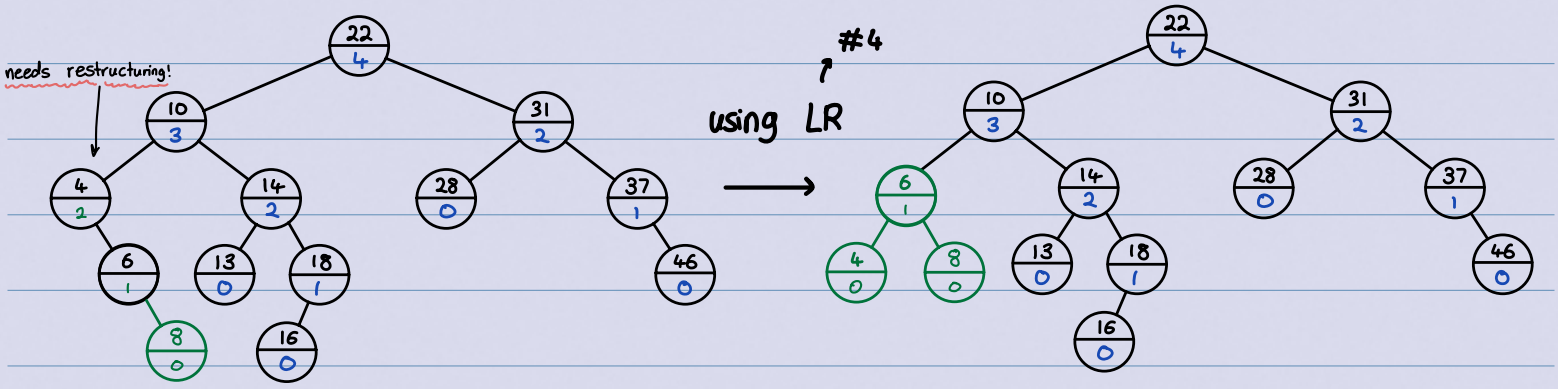
// double-left rotation
rotate-right(c)
u = rotate-left(z)

// left rotation
u = rotate-left(z)

2) return u

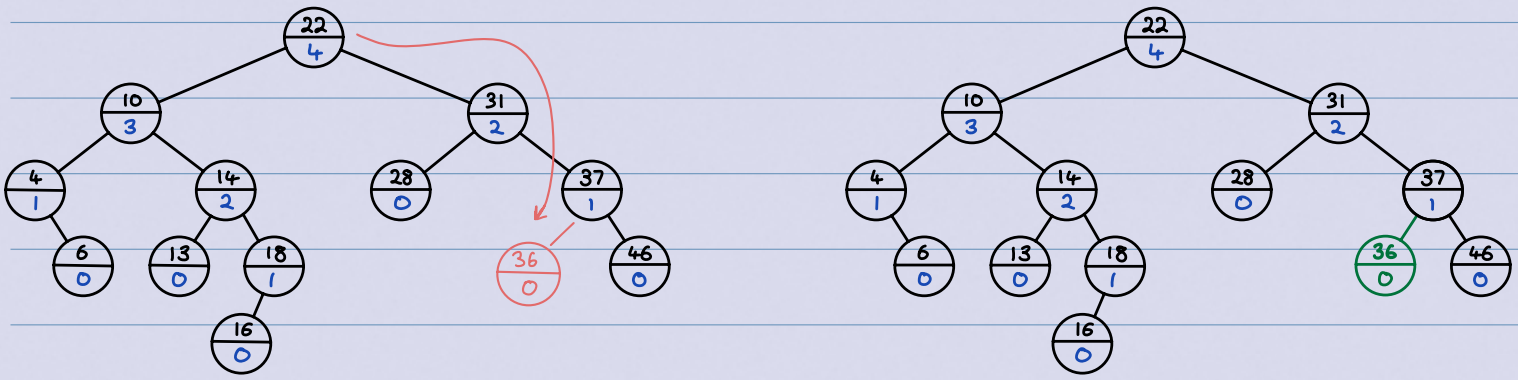
Eg: revisiting AVL::insert(8)

we had previously ended with an unbalanced AVL after inserting 8. Now, we can balance it:

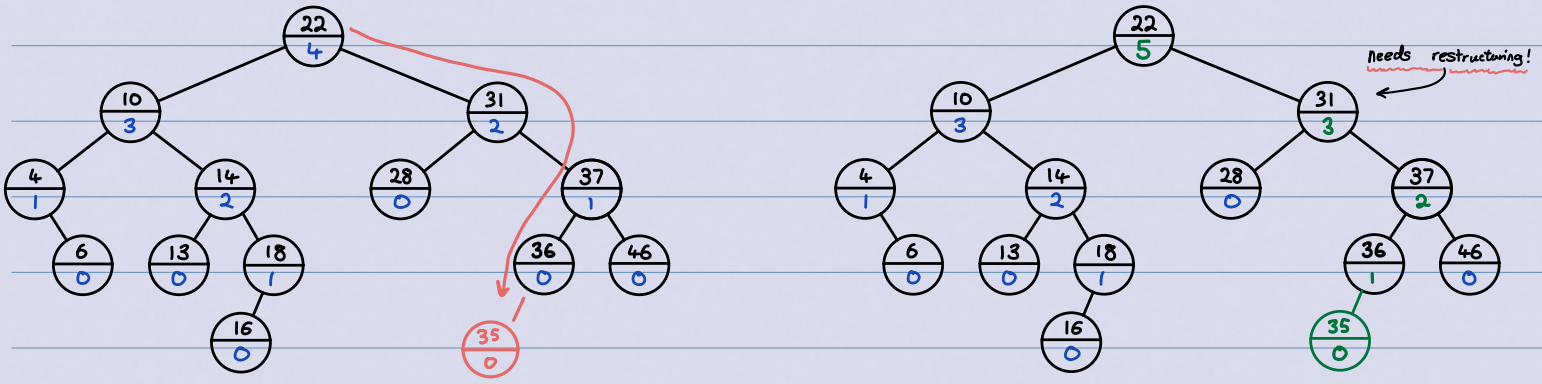


∴, we have correctly restructured the subtree using a left rotation.

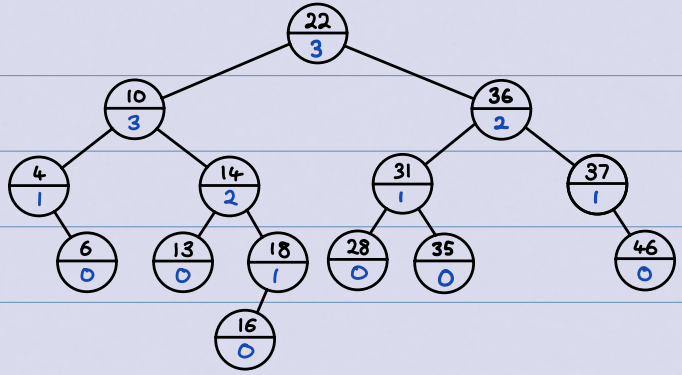
Eg: AVL::insert(35), AVL::insert(36):



→ 35 was inserted and the tree is still correctly structured!



→ 36 was added, but we must restructure, as node 31 is unbalanced! We will restructure node 31 because it's the first unbalanced node as we go up the tree. See that it follows the right-left (31→37, 37→36) pattern, so we must use a double-left rotation:



∴, we have inserted 35 and 36, and restructured accordingly!

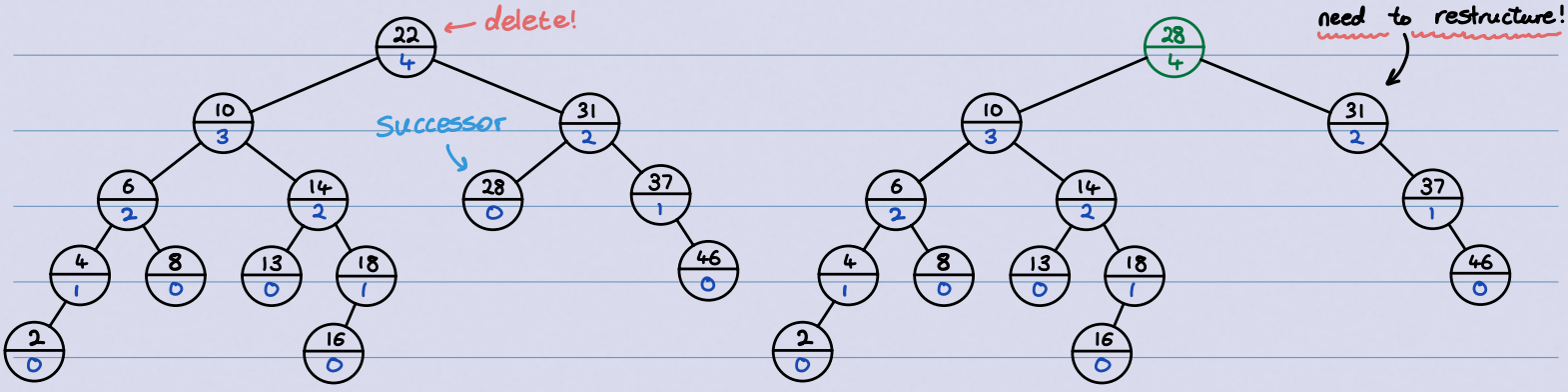
Deletion in AVL Trees

AVL::delete(k) → first, remove the key k with BST::delete. Then, find node where structural change happened (not necessarily near the node that had k!). Go back up to the root, update heights, and rotate if needed.

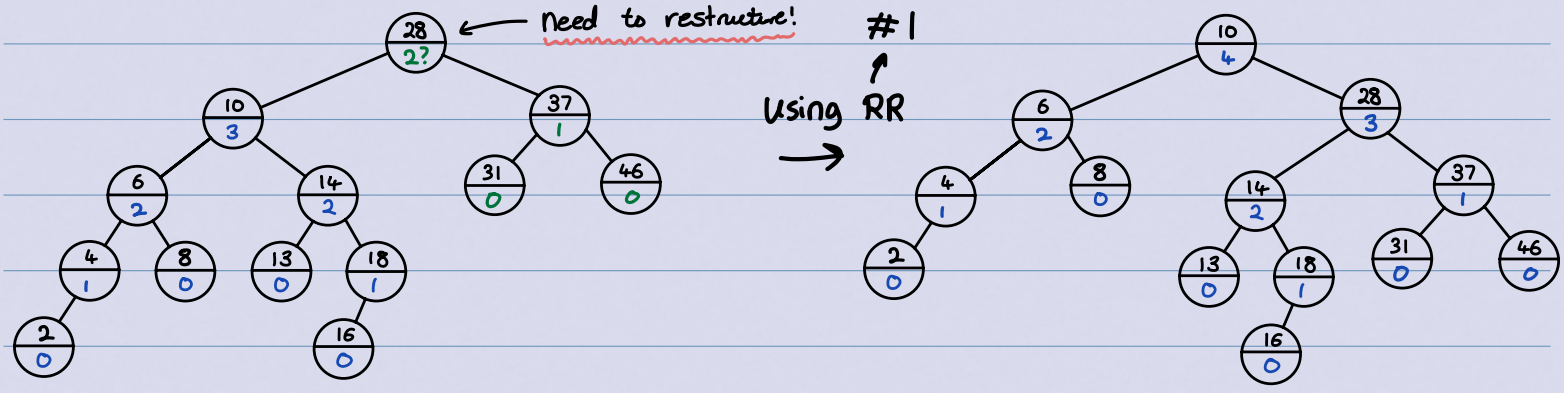
AVL::delete(k)

```
1) z = BST::delete(k)
2) // Assume z is the parent of the BST node that was
      removed
3) while (z is not null) {
4)     if (|z.right.height − z.left.height| > 1) {
5)         c = taller child of z
6)         g = taller child of c   // break ties → avoid double rotation
7)         z = restructure(g, c, z)
8)     }
9)     // always continue up the path
10)    set-height-from-subtrees(z)
11)    z = z.parent
12)}
```

Example: AVL::delete(24):



→ we've deleted the 22 node, but we must now restructure from the 31 node using a left-rotation:



∴, we have deleteded node 22 and restructured accordingly!

- Important: ties must be broken to avoid double rotation while deleting!

AVL Trees - Summary

- Search → just like BSTs, costs $O(height)$
- Insert → BST::insert, the check & update along path to new leaf
  - total cost $O(height)$
  - restructure will be called at most once!
- Delete → BST::delete, then check & update along path to deleted node

- total cost $O(\text{height})$
- restructure may be called $O(\text{height})$ times!

- Worst-case for all operations is $O(\text{height}) = O(\log n)$