Balanced Binary Search Trees

DSA, Fall 2022

Recall — Dynamic Set ADT

```
type Set
type Cell

NewSet:    ()              → *Set
Search:    (*Set, int)     → *Cell
Insert:    (*Set, *Cell)   → ()
Delete:    (*Set, *Cell)   → ()
Minimum:   *Set            → *Cell
Maximum:   *Set            → *Cell
```

Recall — binary search trees

Binary trees where every node has the binary search property:

- all nodes in the left subtree of a node have value less than that node
- all nodes in the right subtree of a node have value greater than that node

To search:

- start at the root, and navigate left or right at every node based on value

To insert:

- start at the root, and navigate left or right at every node based on value
- add new node as a leaf where search would find it

Asymptotic running times

	Linked list	Doubly-linked list	Sorted doubly-linked list (with tail pointer)	Binary search tree
Search	Θ(n)	Θ(n)	Θ(n)	Θ(height(n))
Insert	Θ(1)	Θ(1)	Θ(n)	Θ(height(n))
Delete	Θ(n)	Θ(1)	Θ(1)	Θ(height(n))
Minimum	Θ(n)	Θ(n)	Θ(1)	Θ(height(n))
Maximum	Θ(n)	Θ(n)	Θ(1)	Θ(height(n))

Asymptotic running times

	Linked list	Doubly-linked list	Sorted doubly-linked list (with tail pointer)	Binary search tree
Search	Θ(n)	Θ(n)	Θ(n)	Θ(n)
Insert	Θ(1)	Θ(1)	Θ(n)	Θ(n)
Delete	Θ(n)	Θ(1)	Θ(1)	Θ(n)
Minimum	Θ(n)	Θ(n)	Θ(1)	Θ(n)
Maximum	Θ(n)	Θ(n)	Θ(1)	Θ(n)

Balanced binary search trees

The height of a tree is the number of nodes on the longest path from the root

An empty tree has height 0

A binary tree is balanced if its height is $\Theta(\log_2 n)$

A balanced binary search tree by definition has $\Theta(\log_2 n)$ operations

How do we ensure binary search trees are balanced?

- insert / delete as usual
- repair the tree to restore balance

Balanced binary search trees

The height of a tree is the number of nodes on the longest path from the root

An empty tree has height 0

A binary tree is balanced if its height is $\Theta(\log_2 n)$

A balanced binary search tree by definitio

How do we ensure binary search trees are bal

- insert / delete as usual
- repair the tree to restore balance

This definition is not really actionable

Balanced binary search trees

Many types of balanced binary search trees have been defined:

- AVL trees
- 2-3 trees
- Red-black trees
- Splay trees
- …

Each enforces a specific property that implies that a tree is balanced

AVL trees

An AVL tree is a binary search where every node has the AVL property:

> The difference between the height of the left subtree of a node and the height of the right subtree of a node is at most 1

This implies that AVL trees are balanced

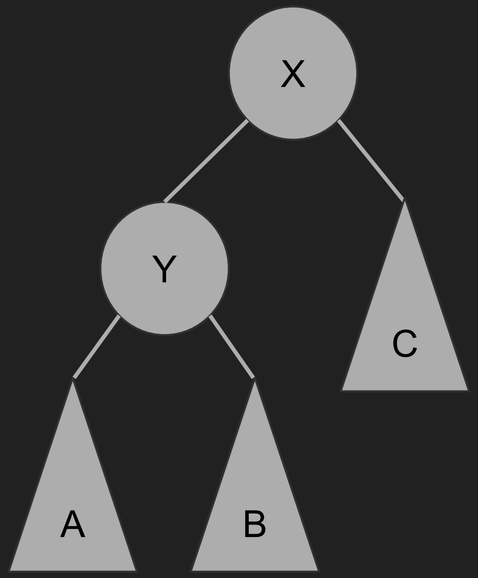
Theorem: An AVL tree of height H has at least $2^{H/2-1}$ nodes

Corollary: The height of an AVL tree with N nodes is at most $2 + 2 \log_2 N$

Thus, the height of an AVL tree is $\Theta(\log_2 n)$

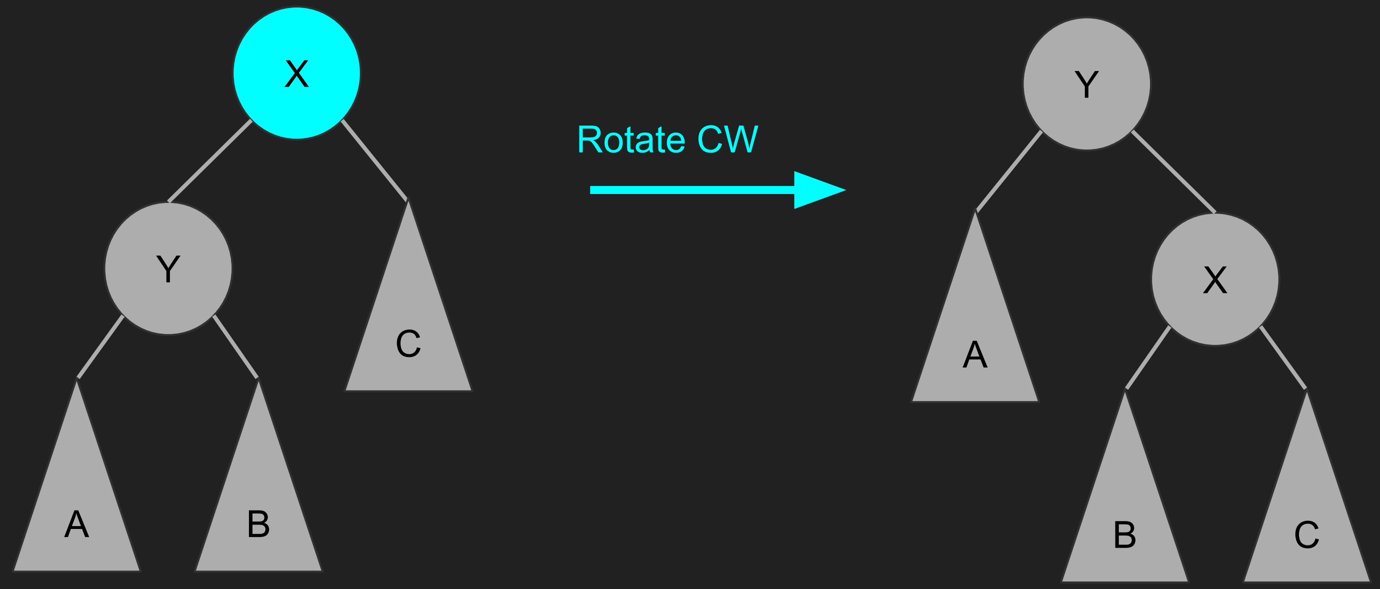
Binary search tree rotations

Rotations that preserve the BST property of all nodes



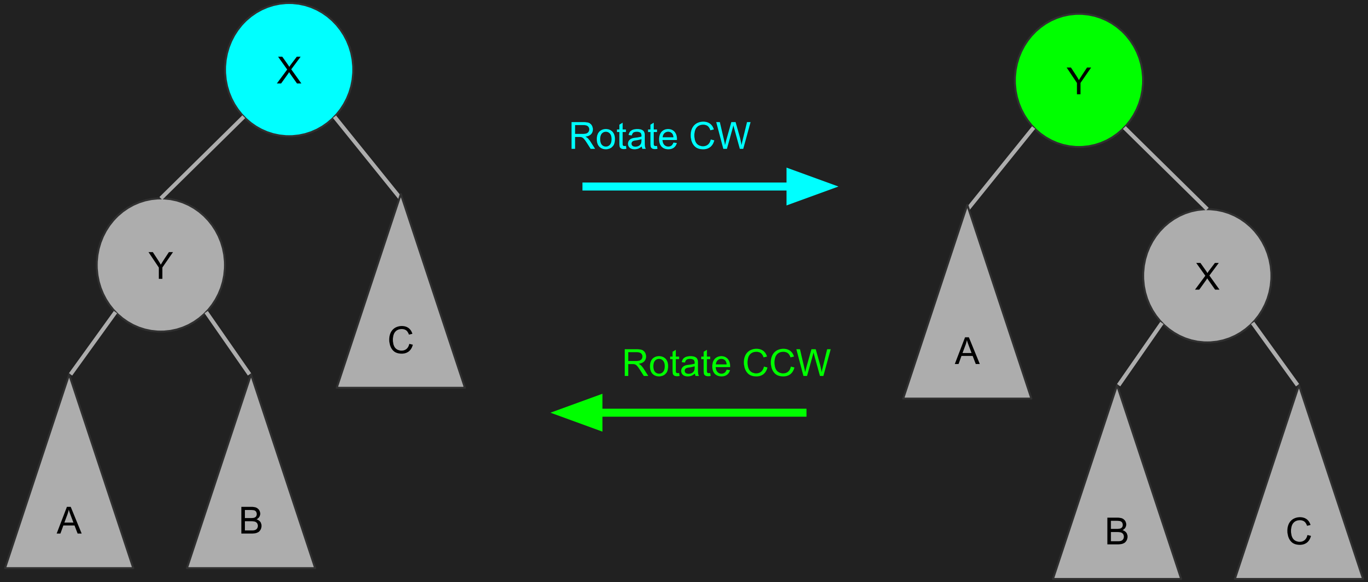
AVL trees — repair transformations

Rotations that preserve the BST property of all nodes



AVL trees — repair transformations

Rotations that preserve the BST property of all nodes



AVL trees — Insert operation

Insert V using the usual binary search tree insertion:

- start at the root
- if V is at the node you're at, you're inserting an existing value — STOP
- if V is less than the value at the node, go to the left child
- if V is more than the value at the node, go to the right child
- repeat until there are no more nodes to follow
- insert a node as the left or right child (depending on V) of the last node visited

Then:

- start at inserted node
- walk back up to the root, repairing the tree on the way up

AVL trees — repairing node X after Insert

Check height of X left and right subtrees

- if difference in height at most 1, continue up
- if difference is 2, repair then continue up

Two cases:

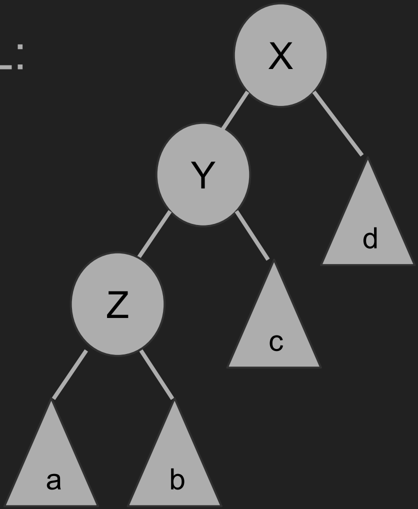
- Case L: highest subtree is the left subtree
- Case R: highest subtree is the right subtree

AVL trees — repairing node X after Insert (case L)

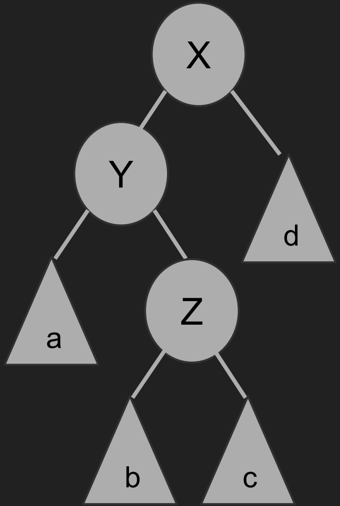
Height of left subtree of X is h + 2, height of right subtree of X is h

So left child of X is a node Y whose height of its subtrees are h and h + 1

Subcase LL:

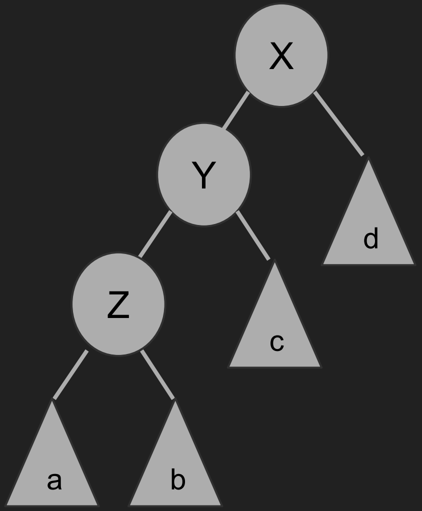


Subcase LR:

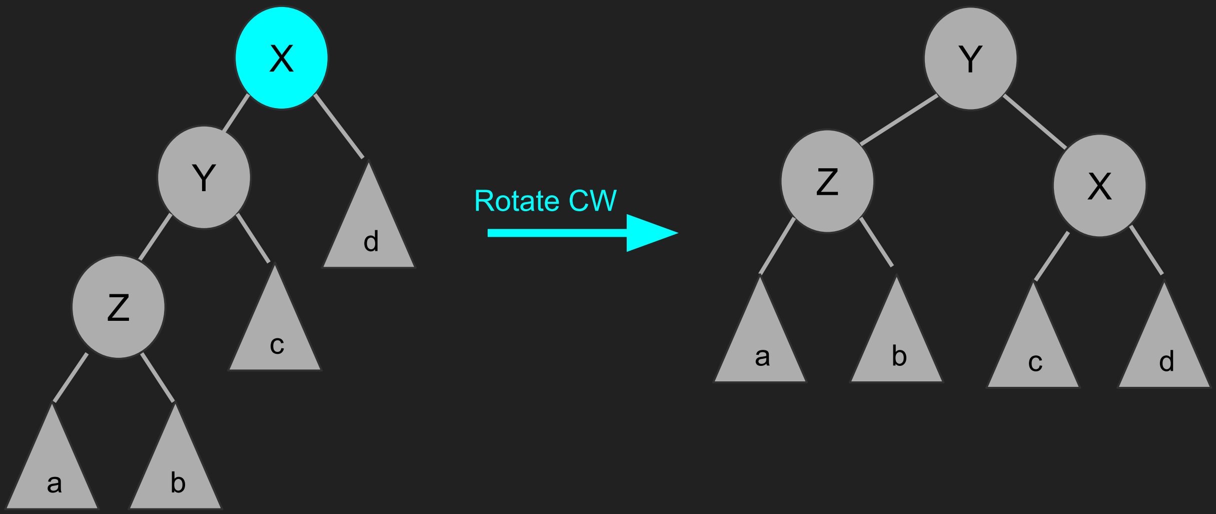


height h

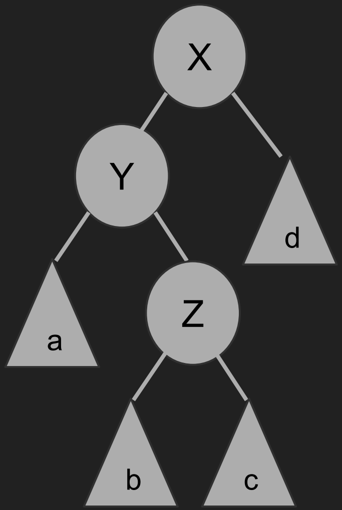
AVL trees — repairing node X after Insert (subcase LL)



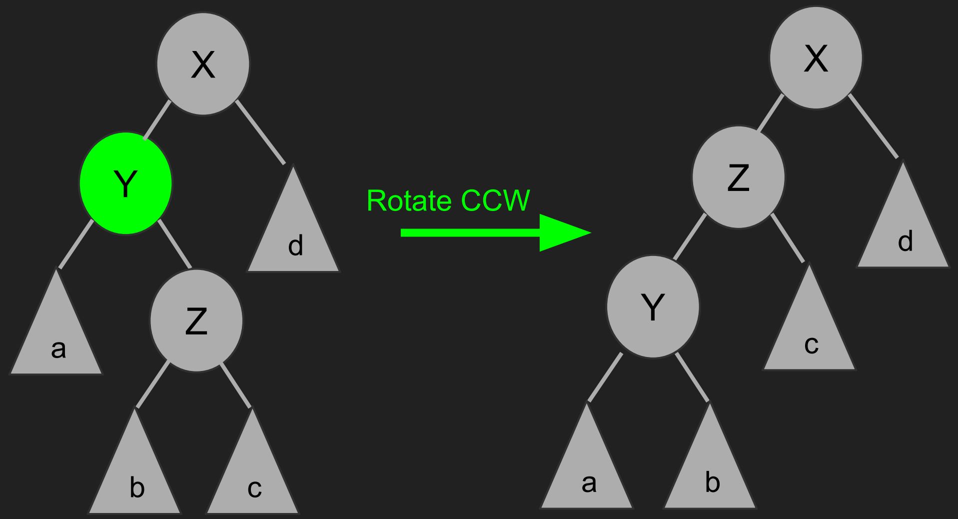
AVL trees — repairing node X after Insert (subcase LL)

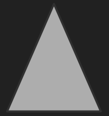


AVL trees — repairing node X after Insert (subcase LR)

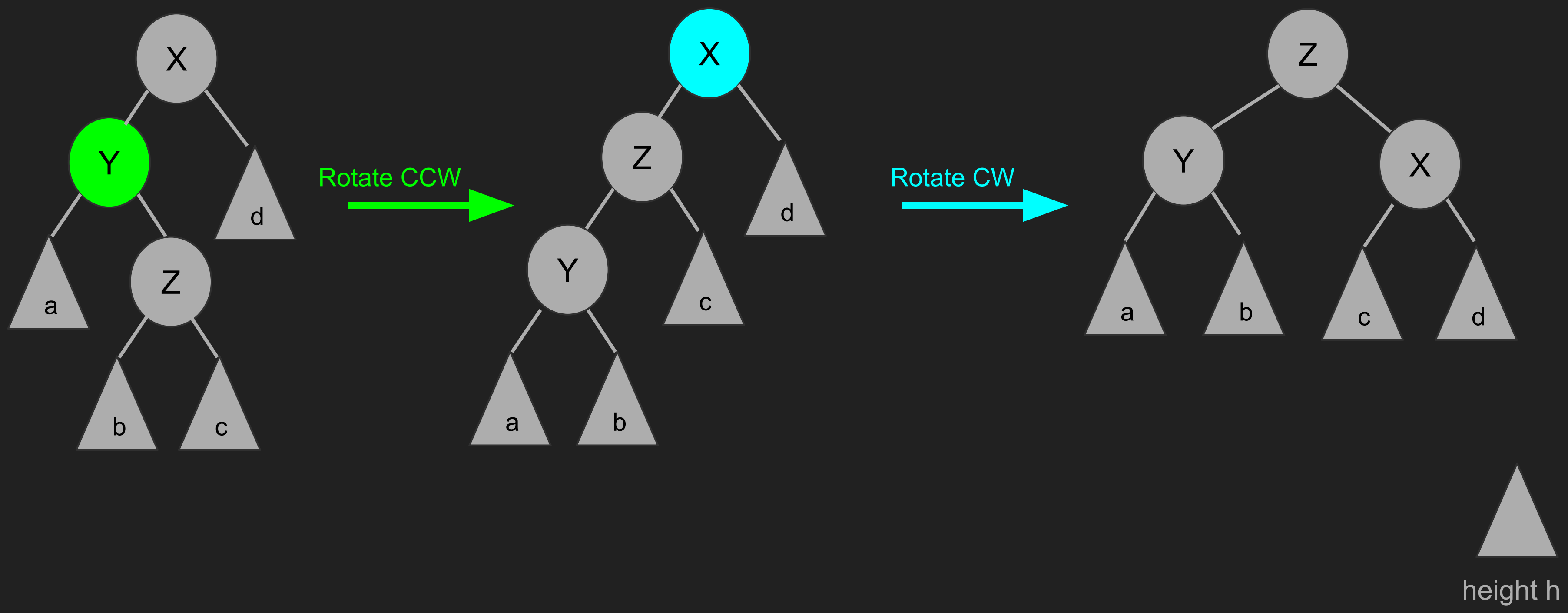


AVL trees — repairing node X after Insert (subcase LR)





AVL trees — repairing node X after Insert (subcase LR)

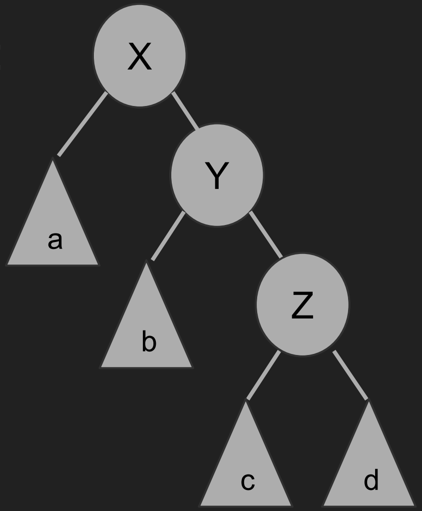


AVL trees — repairing node X after Insert (case R)

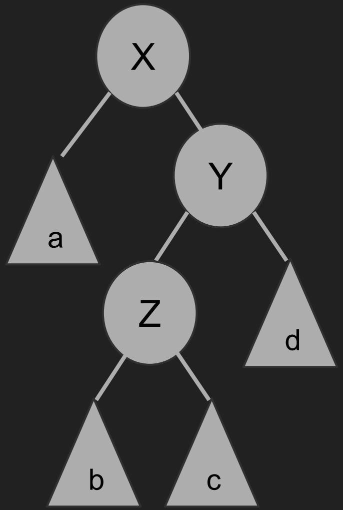
Height of right subtree of X is h + 2, height of left subtree of X is h

So right child of X is node Y whose height of its subtrees are h and h + 1

Subcase RR:

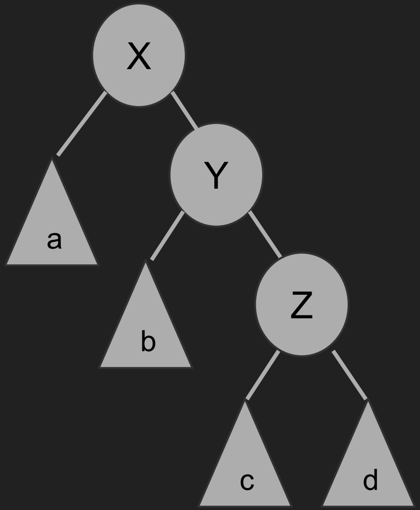


Subcase RL:

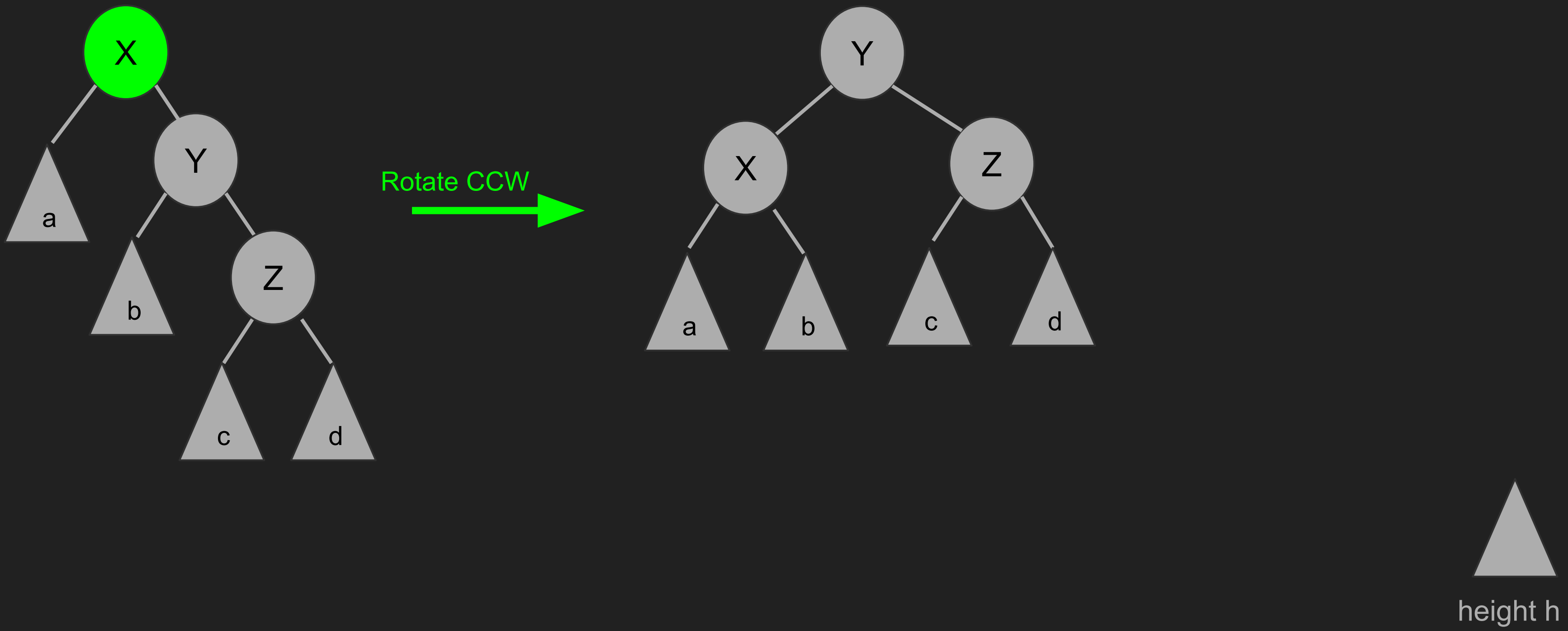


height h

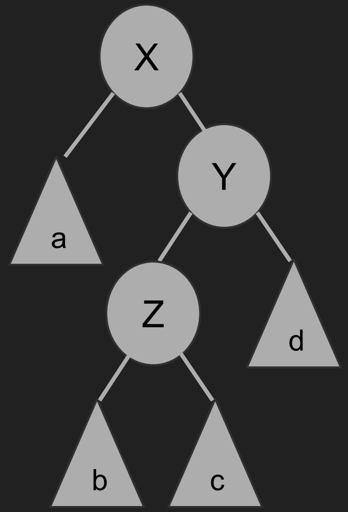
AVL trees — repairing node X after Insert (subcase RR)



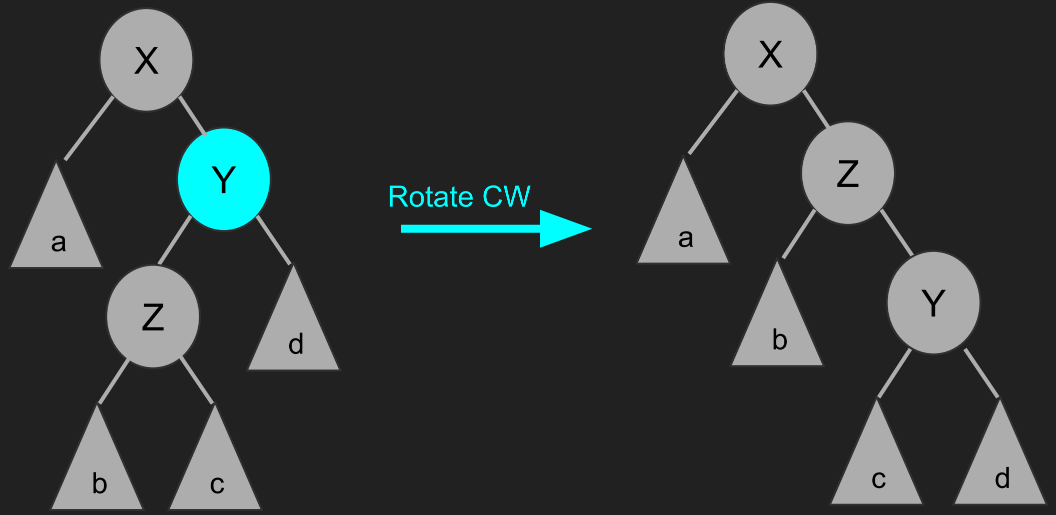
AVL trees — repairing node X after Insert (subcase RR)



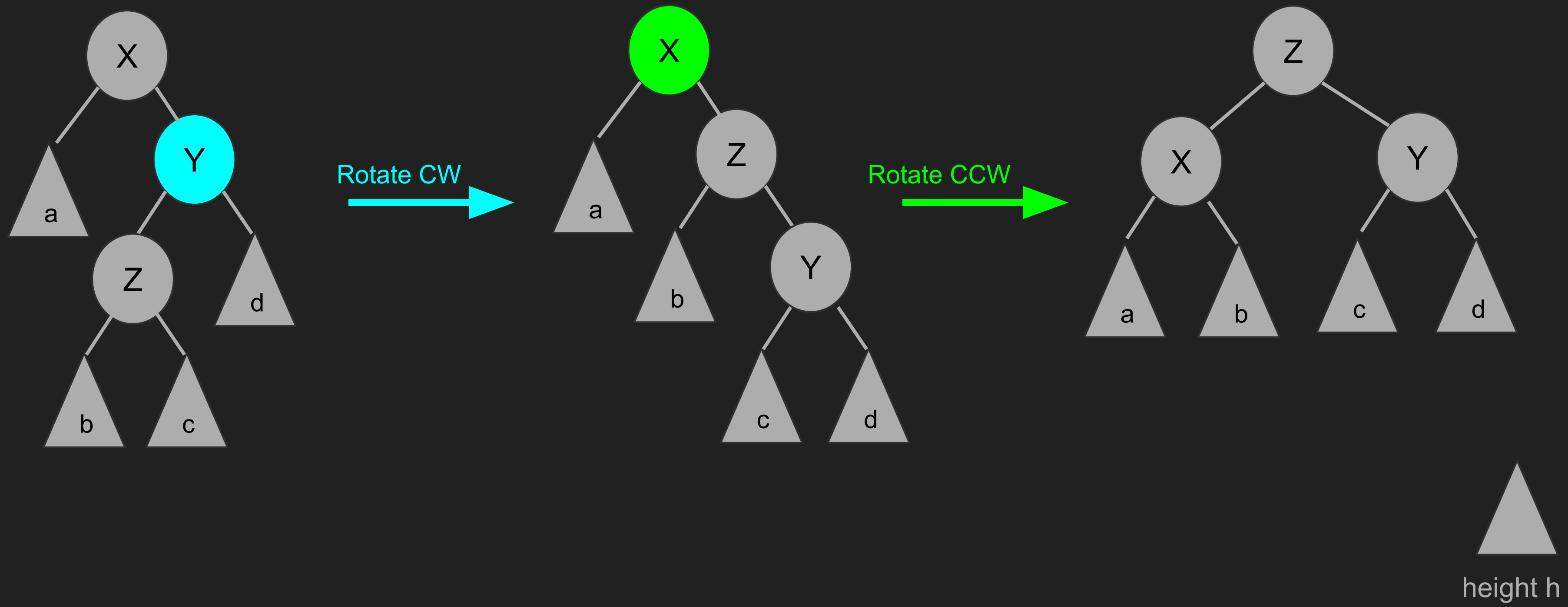
AVL trees — repairing node X after Insert (subcase RL)



AVL trees — repairing node X after Insert (subcase RL)



AVL trees — repairing node X after Insert (subcase RL)



AVL trees — subtlety

You need to compute the height of a node efficiently

Θ(n) if you're not careful

Easiest is to store the height at each node

Update height after insert, delete, and during repairs

```
type Tree struct {
    root *Cell
}

type Cell struct {
    value int
    left *Cell
    right *Cell
    parent *Cell
    height int
}
```

Asymptotic running times

	Sorted doubly-linked list (with tail pointer)	Binary search tree	AVL tree
Search	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n)$
Insert	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n)$
Delete	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$
Minimum	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$
Maximum	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$

Asymptotic running times

	Sorted doubly-linked list (with tail pointer)	Binary search tree	AVL tree
Search	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n)$
Insert	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n)$
Delete	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$
Minimum	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$
Maximum	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$

Making Minimum and Maximum Θ(1)

Store pointers to nodes holding:

- predecessor value
- successor value

at each node

Store pointers to cells with minimum value and maximum values in tree

Update pointers during inserts and deletes

```
type Tree struct {
   root *Cell
   min *Cell
   max *Cell
}

type Cell struct {
    value int
    left *Cell
    right *Cell
    parent *Cell
    pred *Cell
    succ *Cell
    height int
}
```

Asymptotic running times

	Sorted doubly-linked list (with tail pointer)	Binary search tree	AVL tree	AVL tree (with pred / succ)
Search	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Insert	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Delete	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Minimum	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(1)$
Maximum	$\Theta(1)$	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(1)$