AVL Trees. Augmented trees

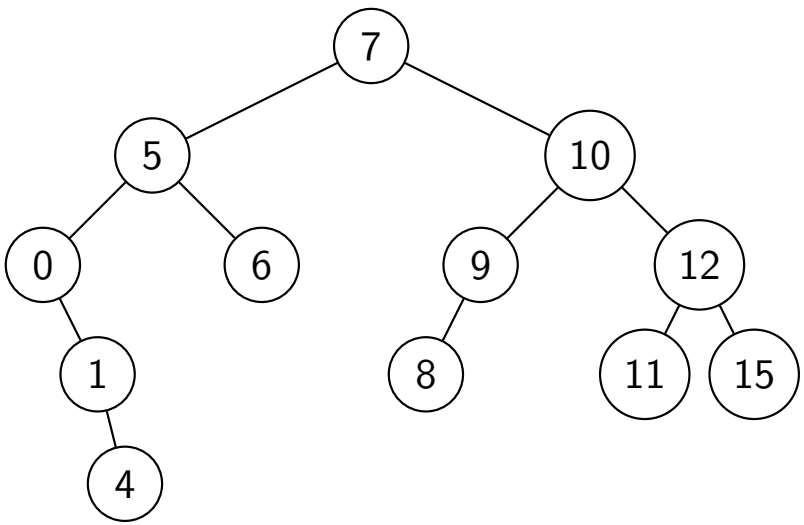
Tufts University

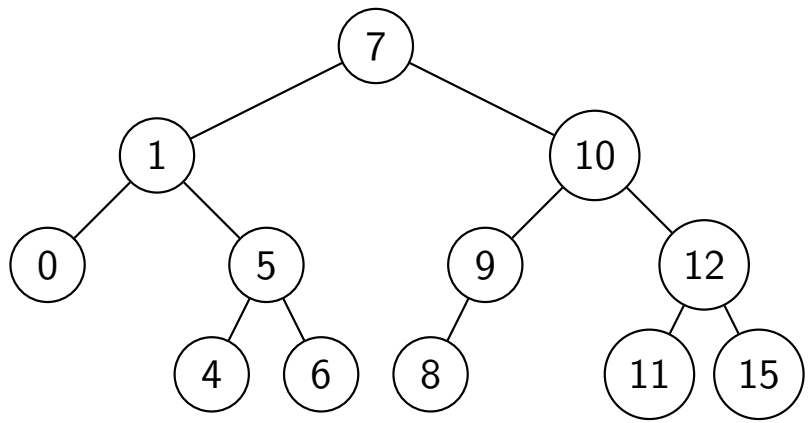
AVL trees - Review

Adelson-Velsky and Landis trees (AVL trees for short) are BST with one more invariant:

- For any node, the difference in heights of subtrees is ≤ 1.

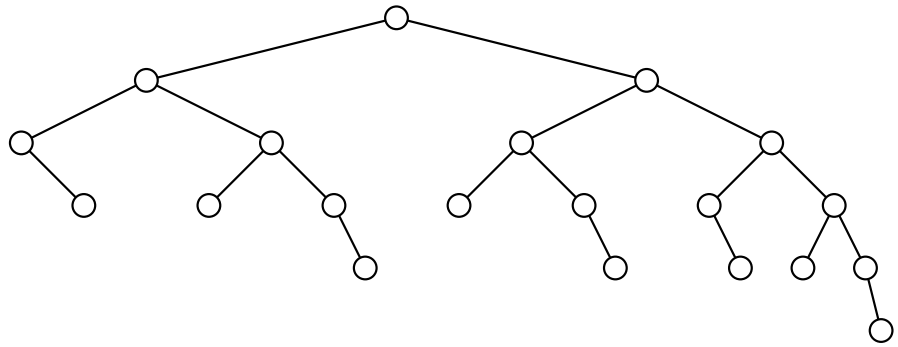
Are these AVL trees?





AVL trees - Height

Lemma The height of an AVL tree is $\Theta(\log n)$.



Proof.

- Let $N(h)$ = minimum # of nodes in AVL tree of height h

  $N(0) = 1$, $N(1) = 2$

  $N(i) = 1 + N(i-1) + N(i-2)$

- $N(h) \geq$ Fibonnacci(h)

  COTD: Fibonnacci(h)$\approx N(h) = \phi^h \approx 1.62^h$

  Alternatively: $N(i) > 2N(i-2) \Rightarrow N(h) \geq 2^{h/2} \Rightarrow$

  $\log N(h) \geq h/2$

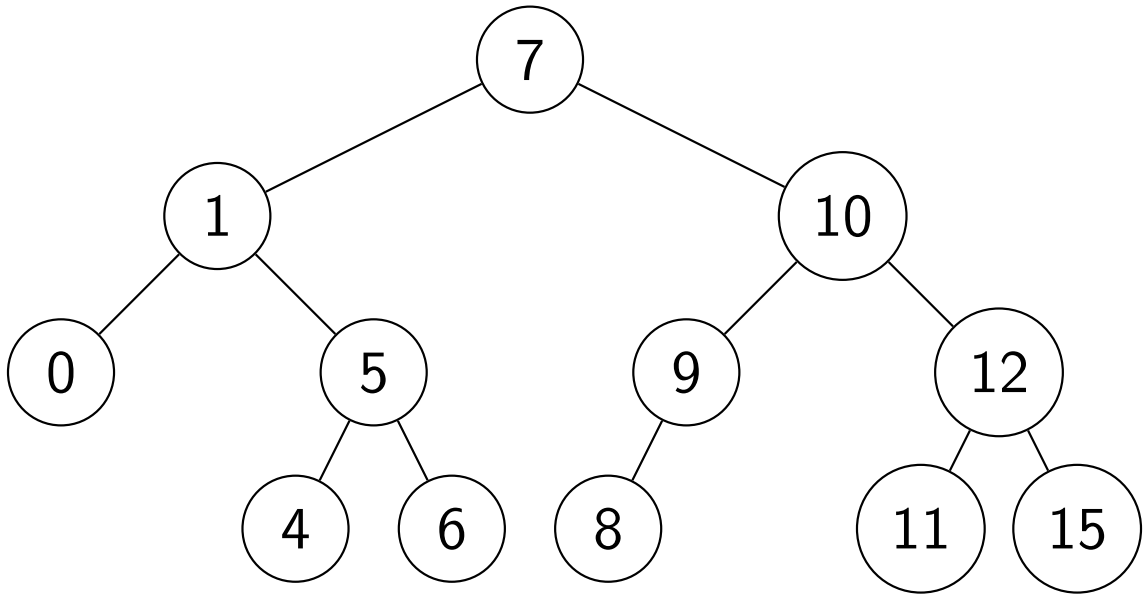
Why do we care?

- Search
- Insert
- Predecessor/Successor
- Find minimum/maximum
- Delete
- ...

ALL in $\Theta(\log n)$!!!

AVL trees - Back to Insert

How do we maintain the AVL rule in Insert?



- BST insertion and if unbalanced rotate
- Yes...but how do we detect if a node is unbalanced?
- We must store extra information at each node

  This is called **augmenting** the tree

Insert pseudocode

AVL-Insert(x, r) \\inserting node x into tree r

BST-Insert(x, r) \\first do a normal BST insert

$x.h \leftarrow 0$ \\set height of x to be 0

$y \leftarrow x.p$ \\y is the parent of x

WHILE $|y.right.h - y.left.h| \leq 1$

$y.h \leftarrow max\{y.right.h, y.left.h\} + 1$ \\update height of y

IF $y = r$ then done

ELSE $y \leftarrow y.p$ \\push y one level up

Balance(y) \\if we exit loop we must balance

Note: trees don't normally store parent.

Exercise: give recursive code (without parent pointer)

Augmented Trees

Data structure with additional information stored to each object.

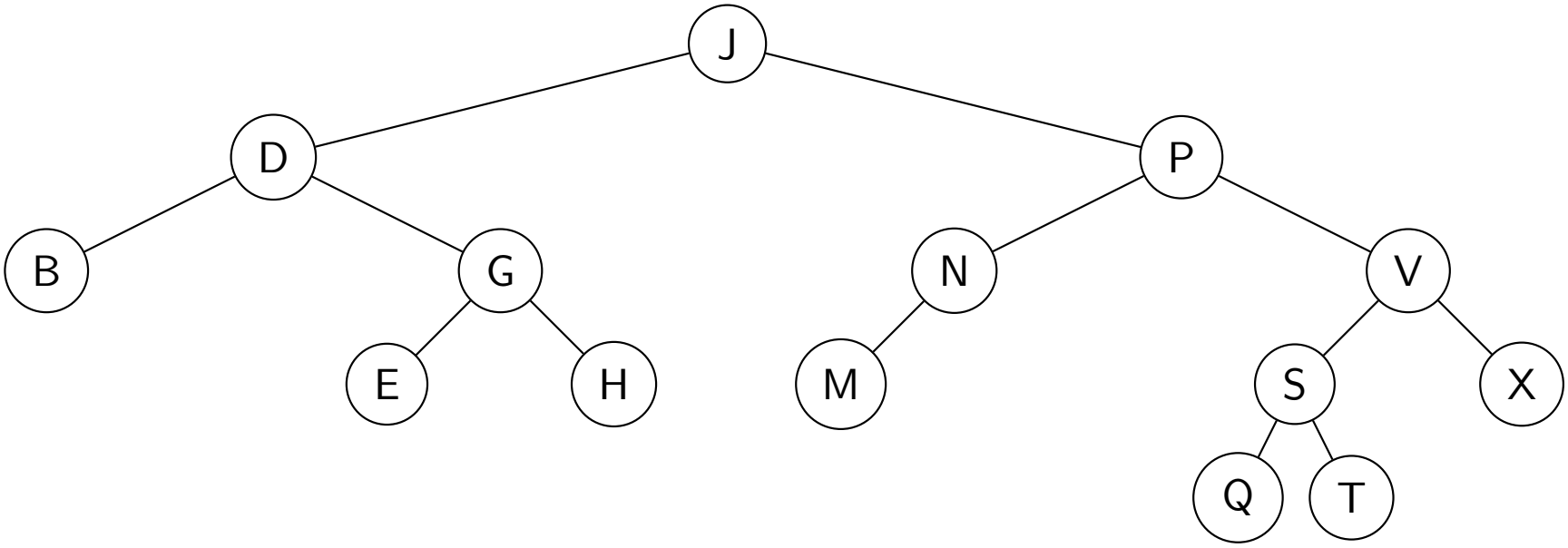
- Information is **position** based

  If you move the object, information should change
- Useful for aggregated data

  Smallest/largest, count, ...
- Main goal: make queries faster
- Great with AVL trees (and many insertions/deletions)
- **Beware**! Standard operations must maintain augmented data

Example 2: Rank-Finding

How can we use augmented trees to do rank-finding?



Rank Q?

Rank-Finding

Options for tree augmentation?

- Option 1: explicitly store rank

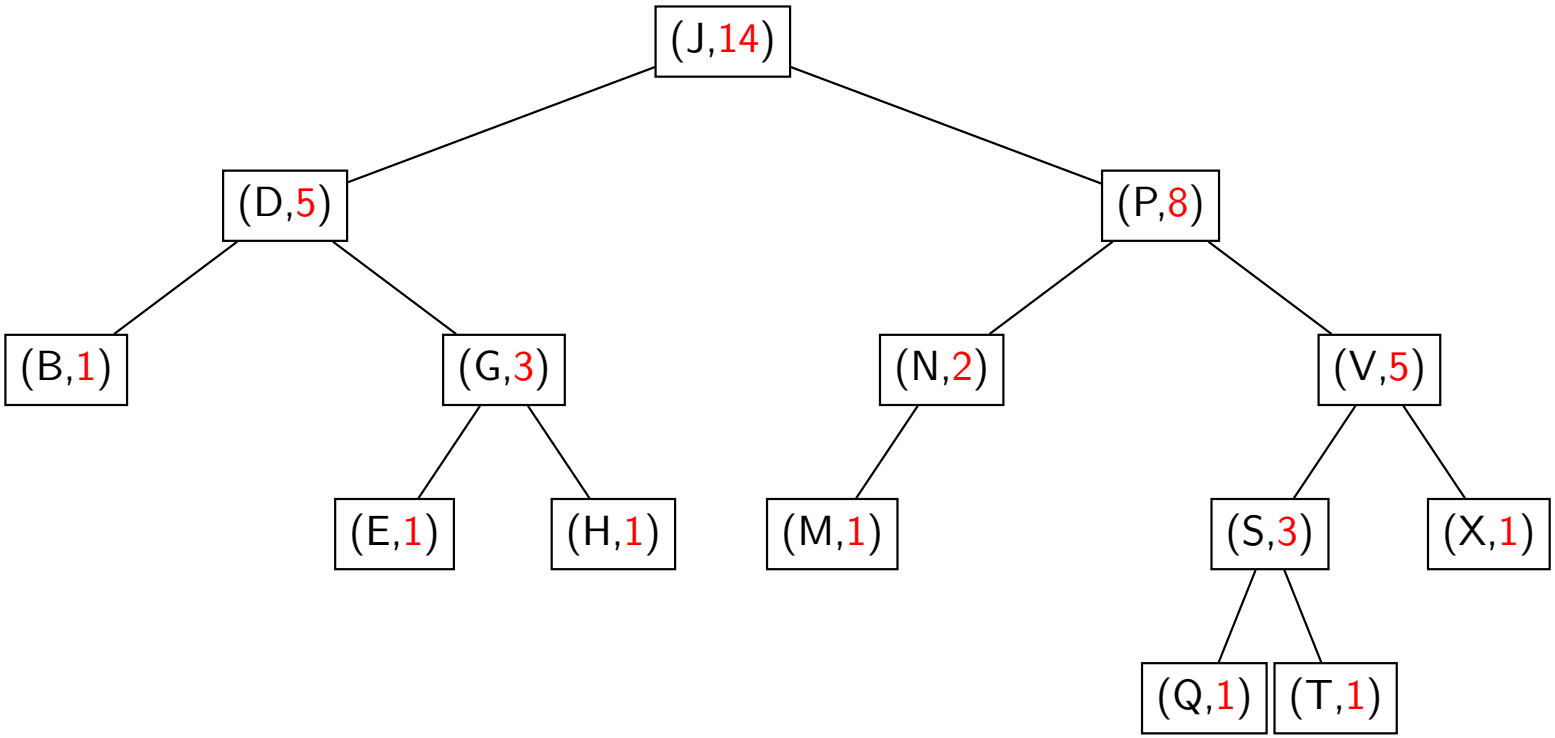
  Queries become easy: search and return augmented info
  Problem: difficult to update

- Option 2: let's store subtree size

  How to do queries?
  What about updates?

Rank-Finding: Query

How can we use augmented trees to do rank-finding?



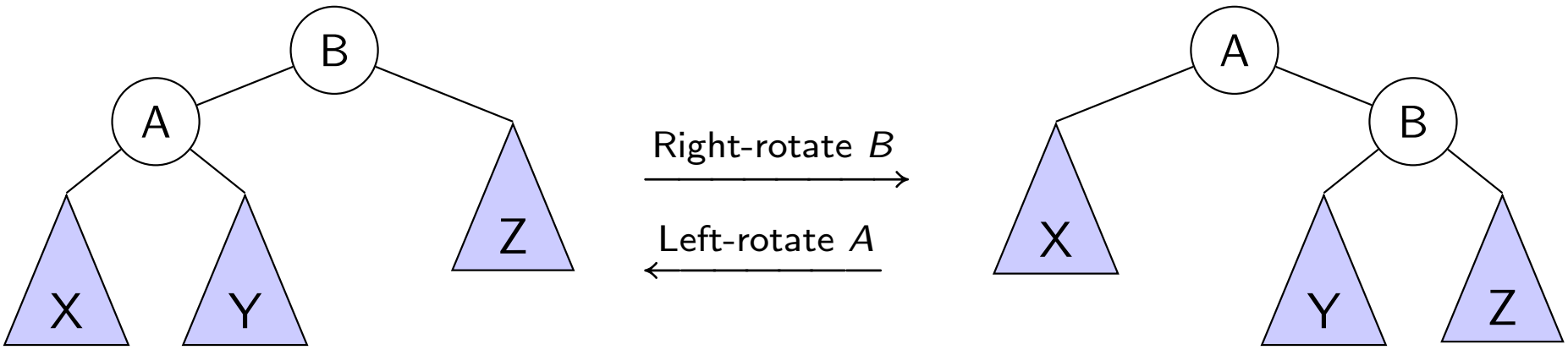
- Use `pos` counter (initially 1)
- If BST search goes left do nothing
- If BST search goes right at *n* increase `pos` by 1+n.left.count
- When target *t* is found return `pos` + t.left.count

Rank-Finding: Insertion

How can we maintain augmented tree during INSERT/DELETE?

Without rotations, only need to increase/decrease count to ancestor along path to root ($\Theta(\log n)$ time).

Insertion/Deletions are both handled with rotations:



Rank-Finding: Summary

Augment each node n by storing `count` (the number of nodes in the subtree induced by n)

Initialization Create empty AVL tree as usual

Rotation When rotating we also locally update `count`

Insertion Insert with modified rotation. Also, must increase count in all traversed nodes

Deletion Delete (also with modified rotation). decrease count in affected nodes

Query Like BST search with `pos` counter.
When going at n right increase `pos` by 1+n.left.`count`
When target t is found return `pos` + t.left.`count`

Lemma

With augmented trees we can handle insertions, deletions and rank finding queries in $\Theta(\log n)$ *time*

Summary

- Augmented trees are a powerful new tool
- Store information based on current location

  Move the node $\Rightarrow$ augmented information changes
  Information maintained under insertion/deletions

  - Need to find right attribute to augment
  - More examples in next lecture!

Additional practice questions

- Modify augmentation to do selection? (i.e. return the object with rank k).
- Compare augment tree with sorted/unsorted array
  - Runtime of Insertion? Deletion? Query?
- What about hash tables?