LINKED STRUCTURES IN JAVA (#3)

Prof. Chris Jermaine
cmj4@cs.rice.edu

In A6, We Are Implementing a Top-K Alg

- Idea: have a class we can add many (score, data) pairs into
  - — Then we can ask the structure for the K “data” object with lowest scores
  - — Assume that K is known at structure initialization
  - — But the number of pairs to process is not known
- Should be fast!
  - — At most O(log K) cost to process each pair

In A6, We Are Implementing a Top-K Alg

- Can we use a BST to do this?
  - — Add first K pairs directly into a BST
  - — Build the tree using the score
  - — When have K + 1 pairs, remove the largest from the tree and discard
- Will be fast
  - — If can guarantee O(log K) insertion time into tree with K items
  - — If can guarantee O(log K) time to delete the pair with the largest score
- Unfortunately, simple BST won’t cut it...

The Simple BST Can Have Bad Performance

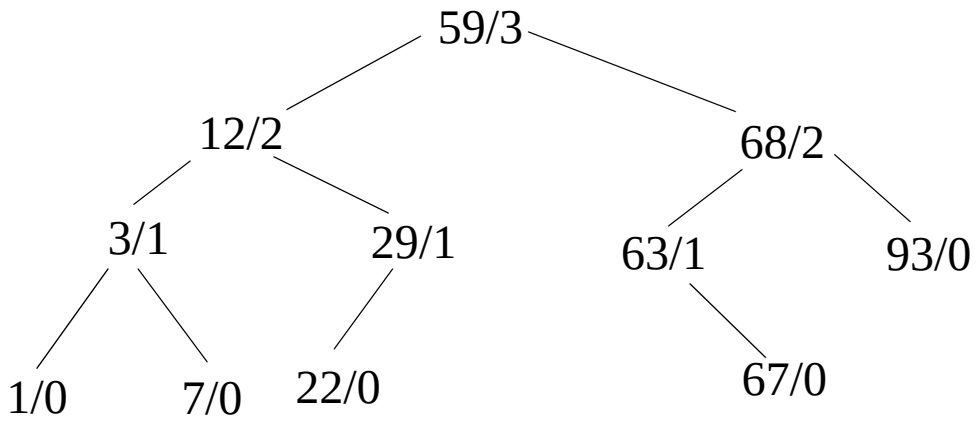
- May become unbalanced under certain insert orders
  - — Sorted, in-order or reverse-order inserts, for example
  - — In our application, this might happen
- The fix: are many self-balancing BST variants
  - — These guarantee good performance under any insert order
  - — Two most famous are AVL trees and Red/Black trees

AVL Trees

- Height of subtree is stored in every node
  - — Is the longest distance from root of subtree to leaf
- Guaranteed to be height-balanced
  - — |height_left - height_right| < 2 at every node in the tree

AVL Trees

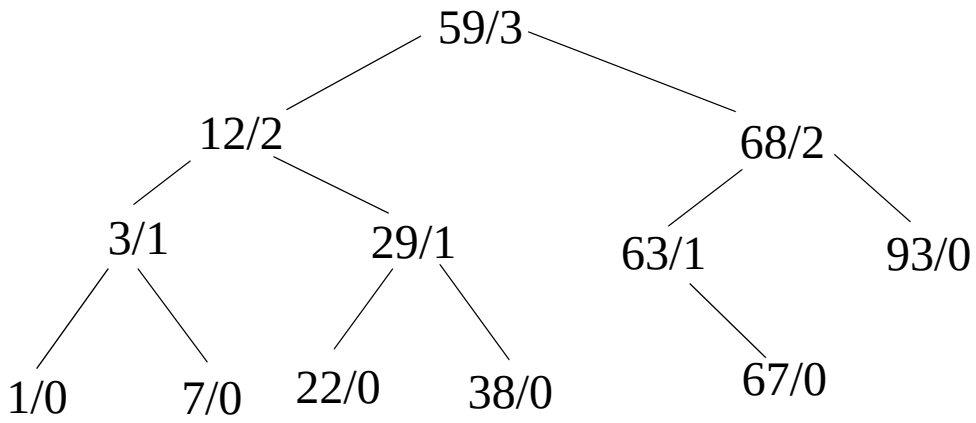
- Height of subtree is stored in every node
  - — Is the longest distance from root of subtree to leaf
- Guaranteed to be height-balanced
  - — |height_left - height_right| < 2 at every node in the tree



For each node we have key/height

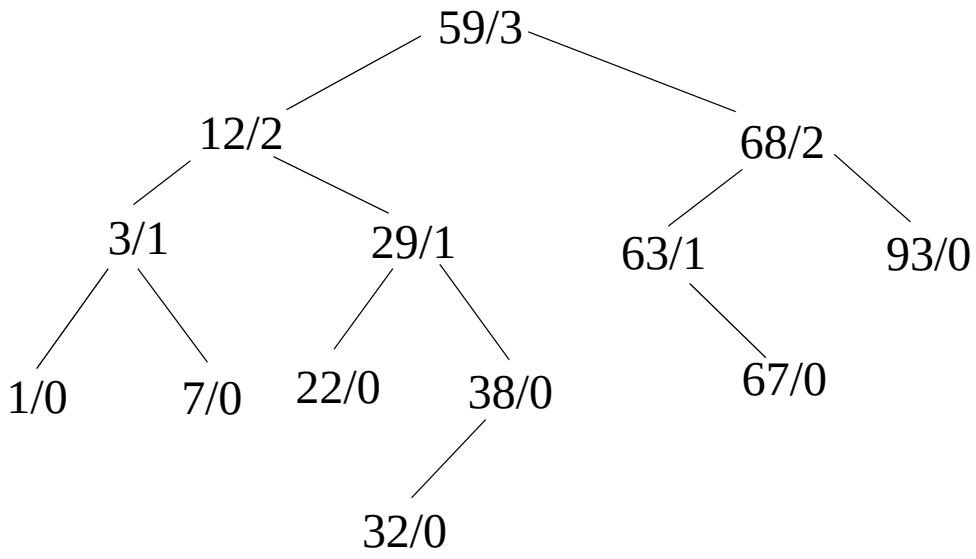
AVL Inserts

- Inserts are easy as long as tree remains balanced
  - — Not much different from inserts into non-self-balancing BST
- Example: add 38



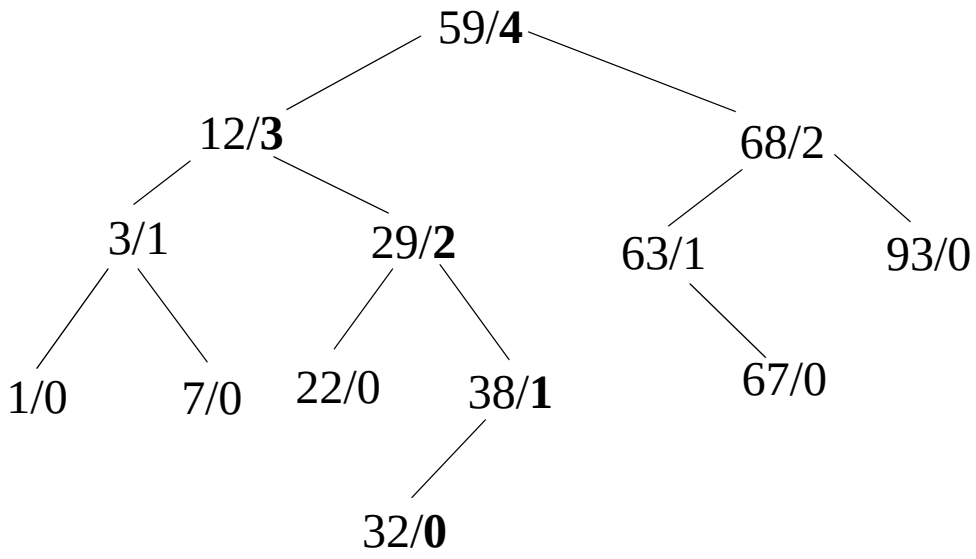
AVL Inserts

- Inserts are easy as long as tree remains balanced
  - — Not much different from inserts into non-self-balancing BST
- Example: add 38, then 32



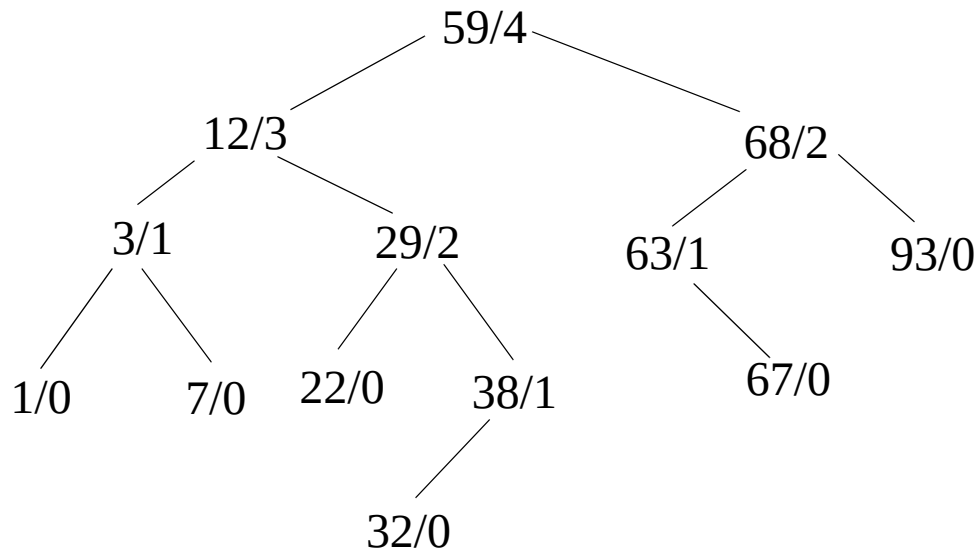
AVL Inserts

- Inserts are easy as long as tree remains balanced
  - — Not much different from inserts into non-self-balancing BST
- Example: add 38, then 32
  - — Note: Possibly to update height after you recursively insert!



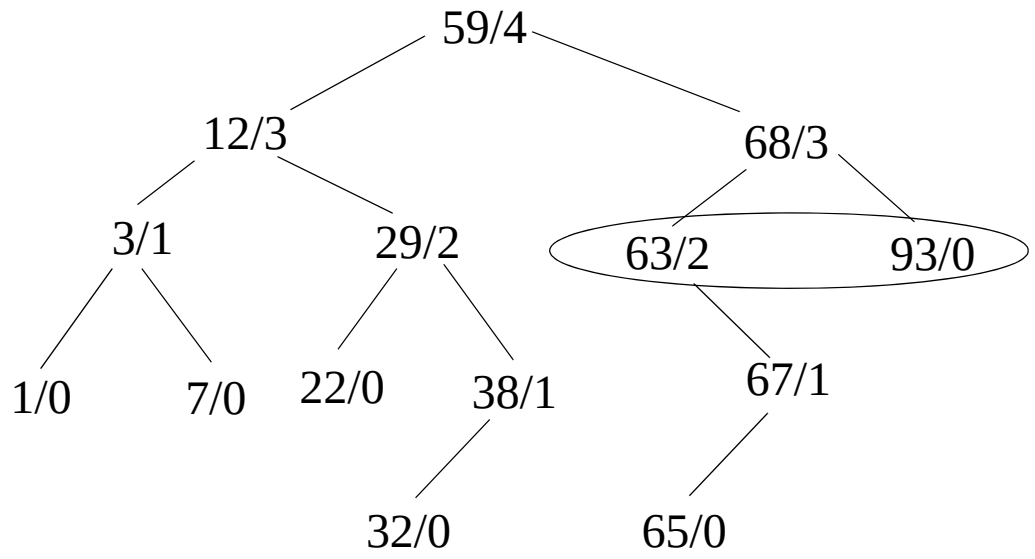
AVL Inserts: Regaining Balance

- But what if we become unbalanced?
  - Example: insert a 65?



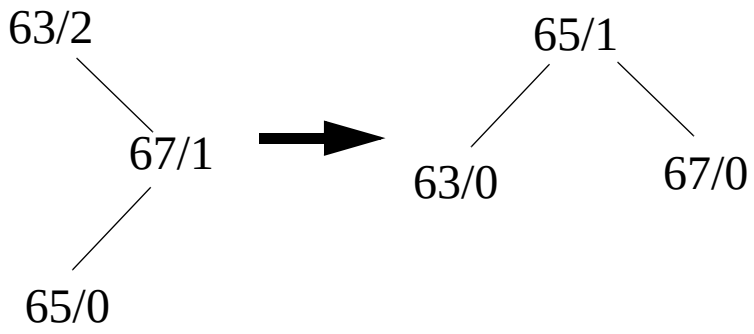
AVL Inserts: Regaining Balance

- But what if we become unbalanced?
  - — Example: insert a 65?
  - — We know it is unbalanced cause after recursive insert
  - — Heights of two children differ by 2



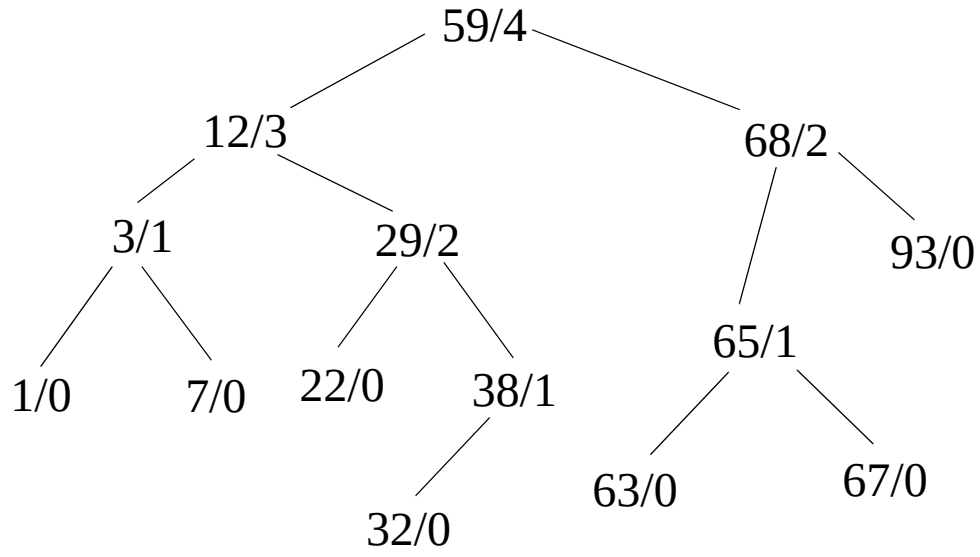
Rotation

- Handled via a **rotation**
  - — Rotation is used in all self-balancing BSTs to re-balance
  - — Are many flavors of rotation



Rotation

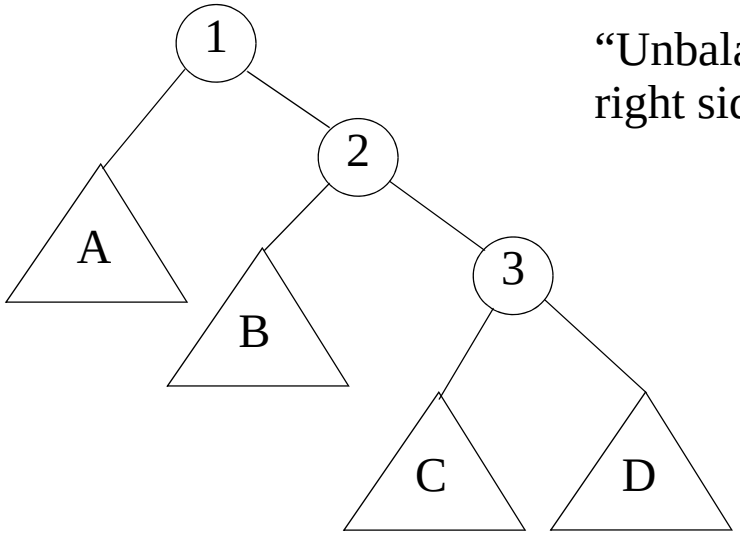
- After rotation, we have a balanced tree once again!
- Note: Basic idea of rotation is easy
  - — But understanding all of the cases/types of rotation takes more effort
  - — In fact, are four types of rotations needed to handle inserts
  - — We will go through them systematically



Rotations in AVL Trees

- Two types of rotations are easy
  - — Unbalanced to right (right deep)
  - — Unbalanced to left (left deep)
- Let’s look at both of them now

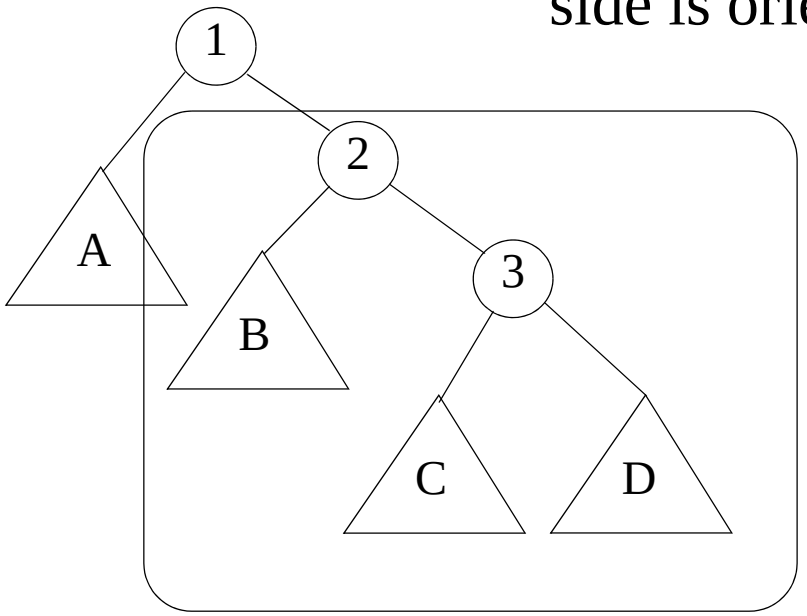
Unbalanced to right (right deep)



"Unbalanced to right" since right side is deeper

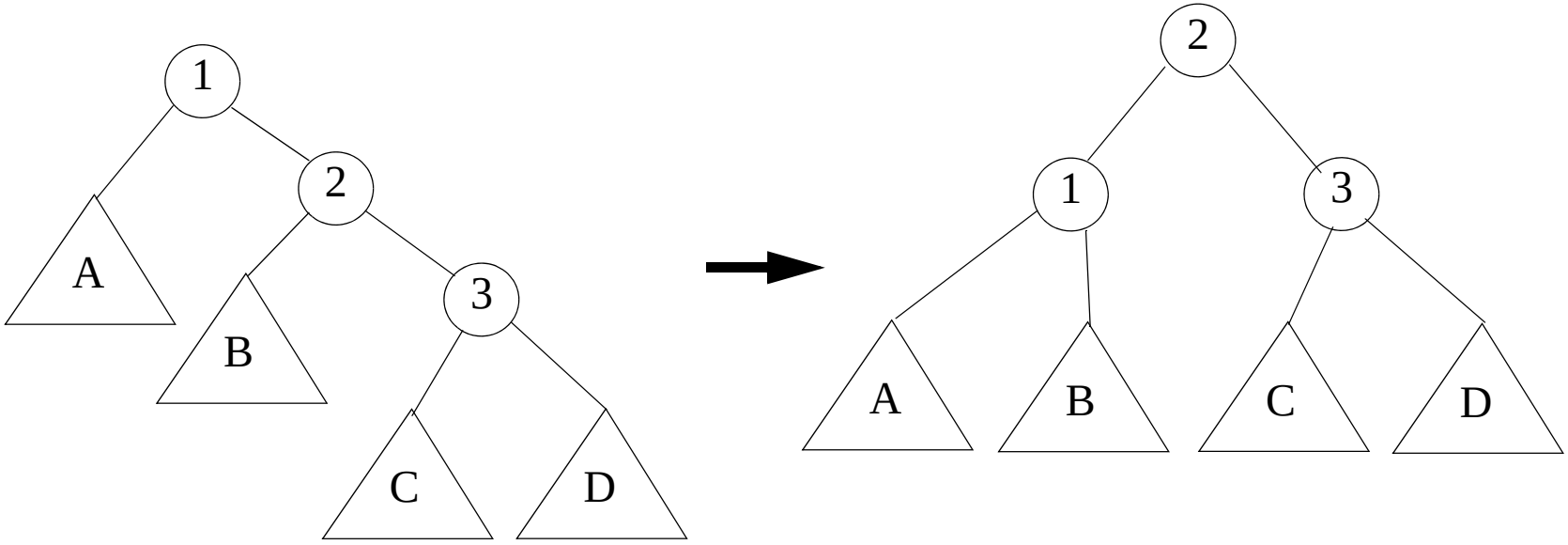
Unbalanced to right (right deep)

"right deep" since unbalanced side is oriented to the right



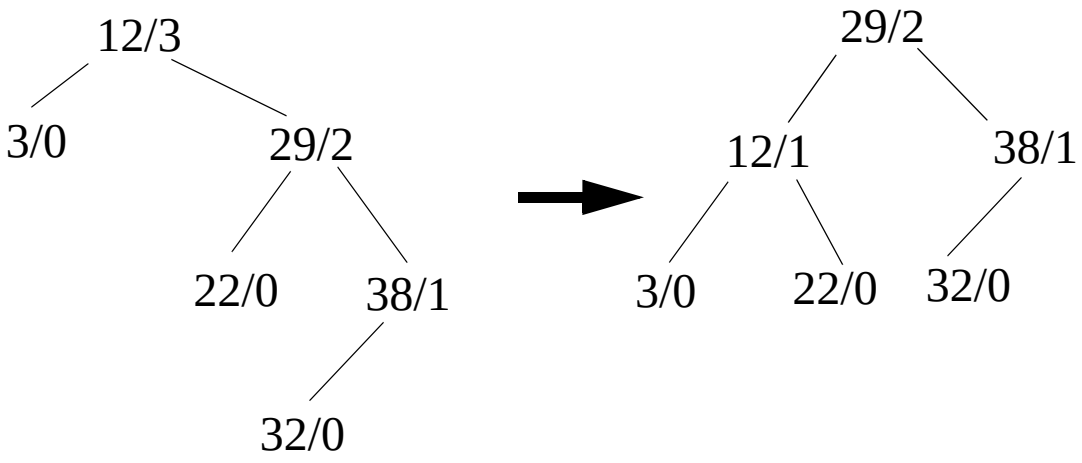
Unbalanced to right (right deep)

- The rotation to re-balance this is easy!



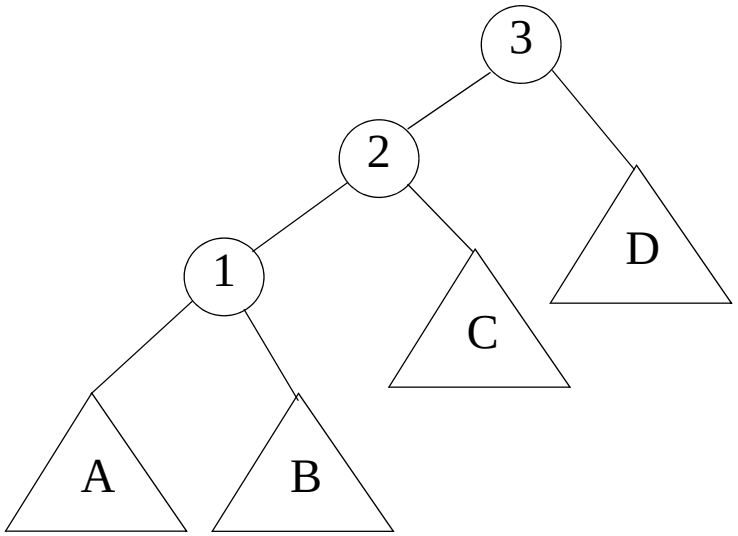
Unbalanced to right (right deep)

- Here is a concrete example



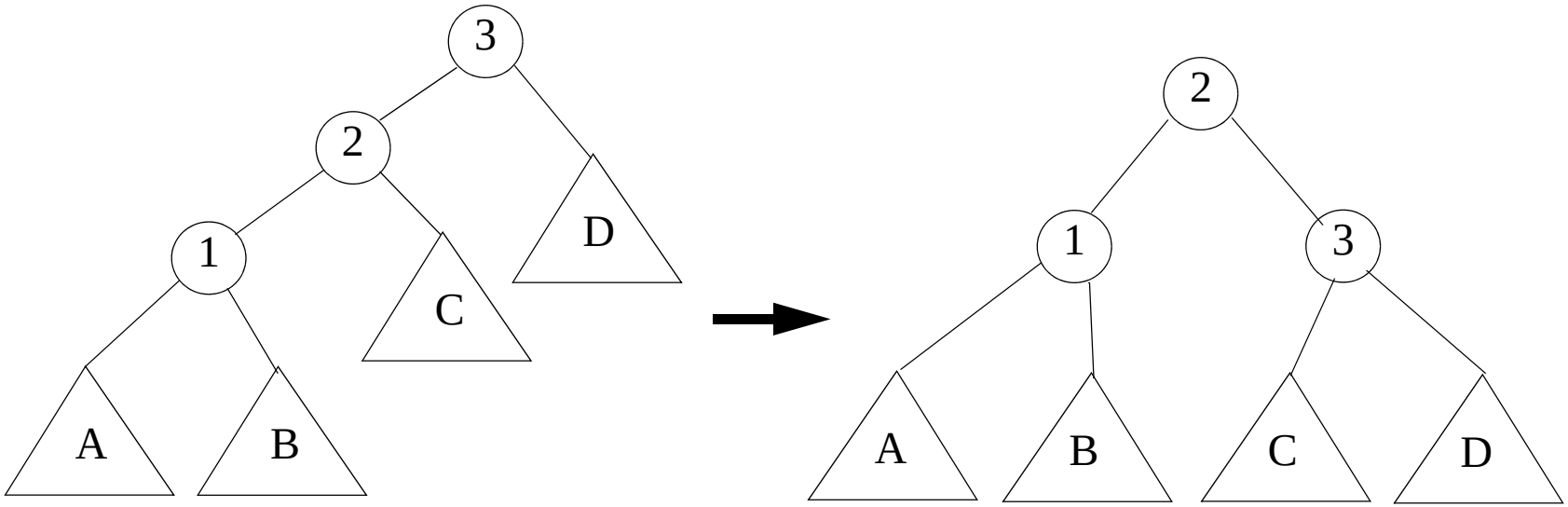
Unbalanced to left (left deep)

- This one is similarly easy



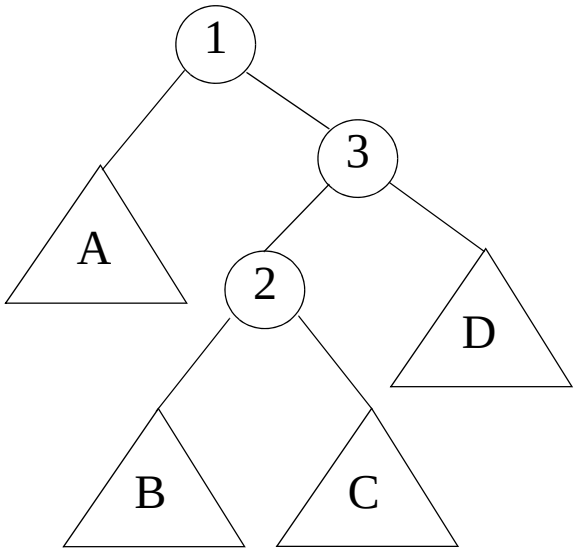
Unbalanced to left (left deep)

- This one is similarly easy



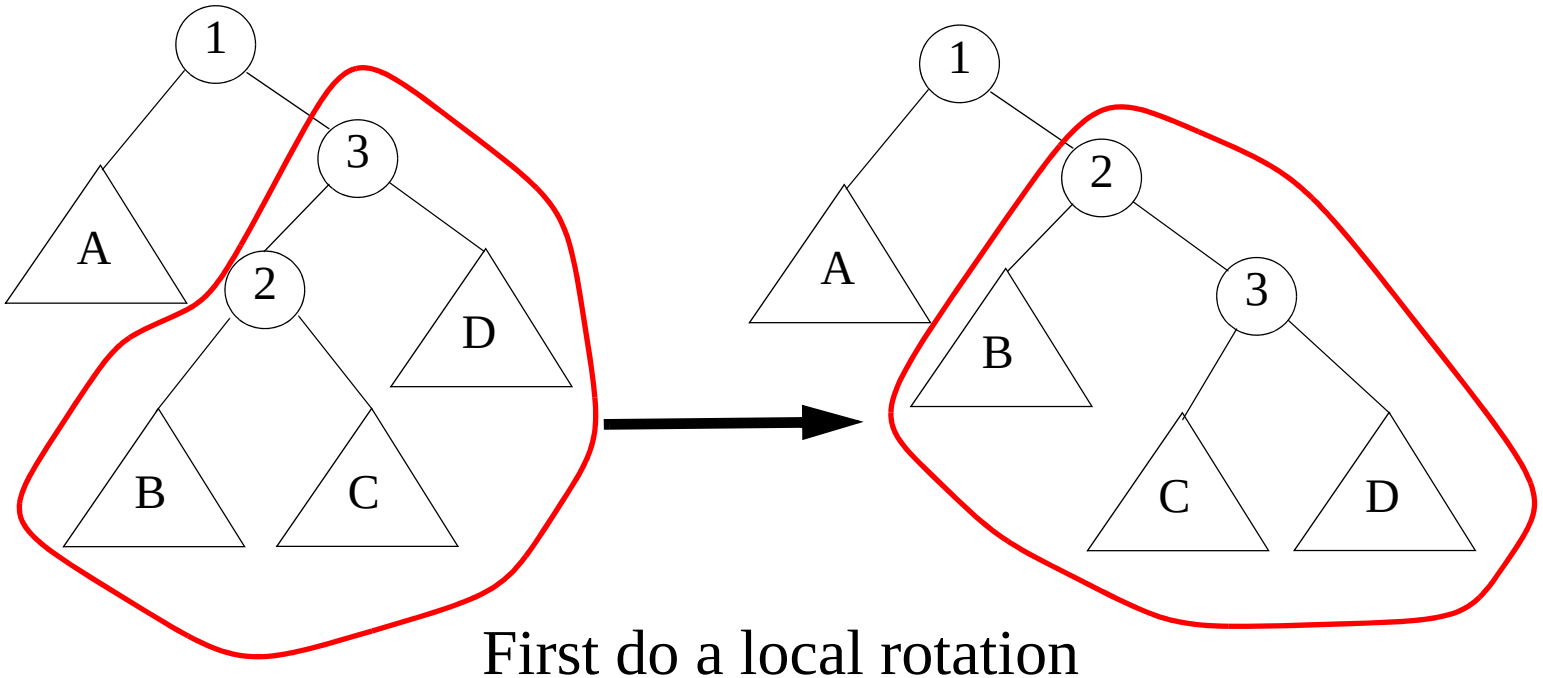
Unbalanced to right (left deep)

- However, two of the rotations require more thought
  - — Fortunately, they are not too difficult!



Unbalanced to right (left deep)

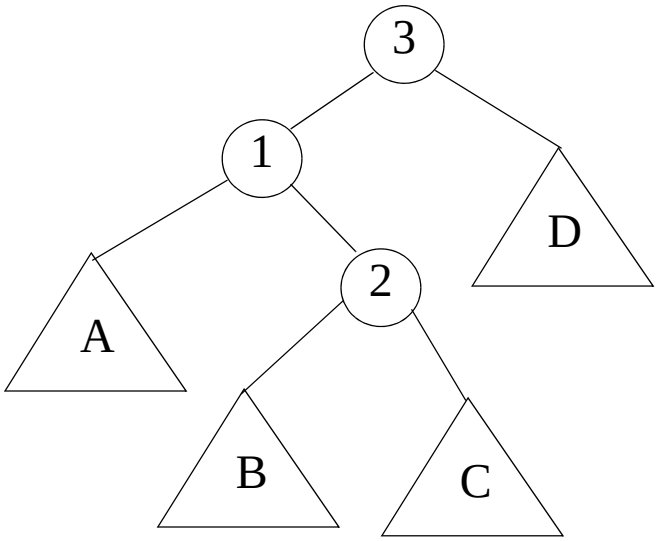
- However, two of the rotations require more thought
  - — Fortunately, they are not too difficult!



First do a local rotation
This gives you a right-deep subtree
Now we can handle this!

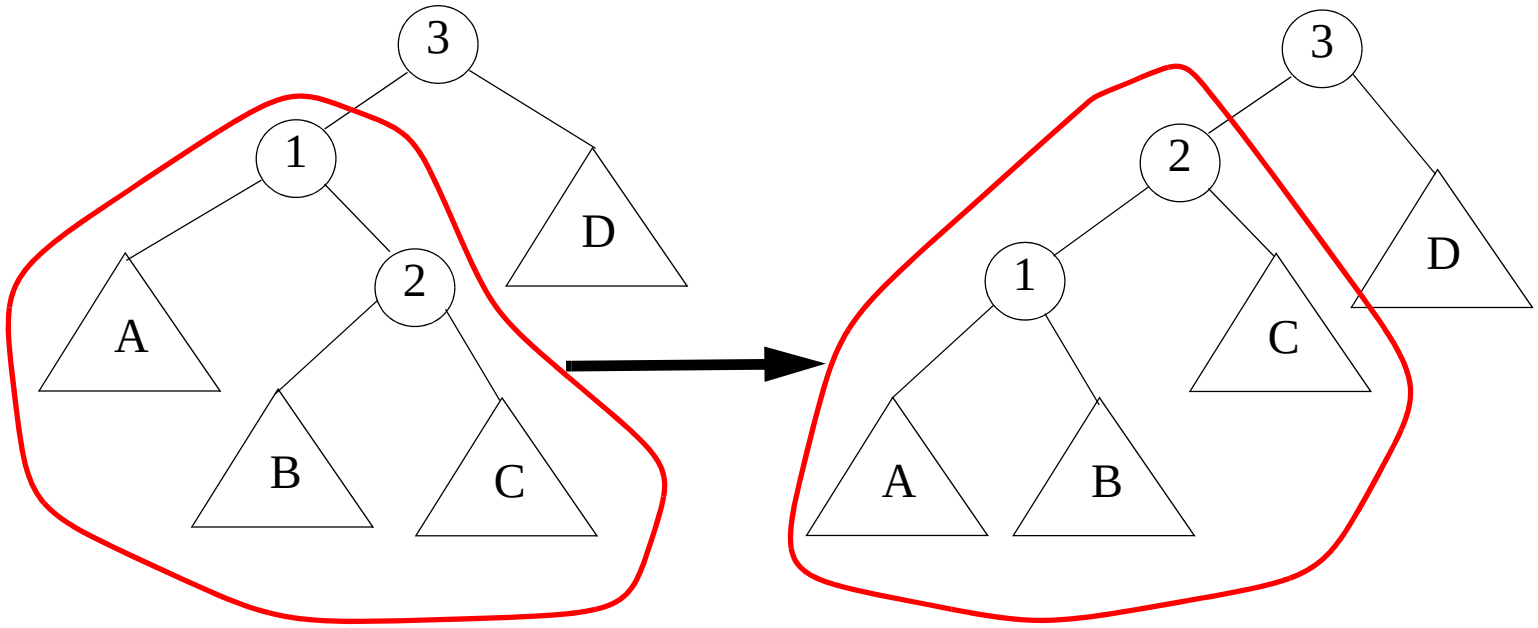
Unbalanced to left (right deep)

- This case is very similar



Unbalanced to left (right deep)

- This case is very similar



Again, do a local rotation to make the sub-tree left deep
Then we can handle it!

Putting It Together

- So what does final insert pseudo-code at a node look like?

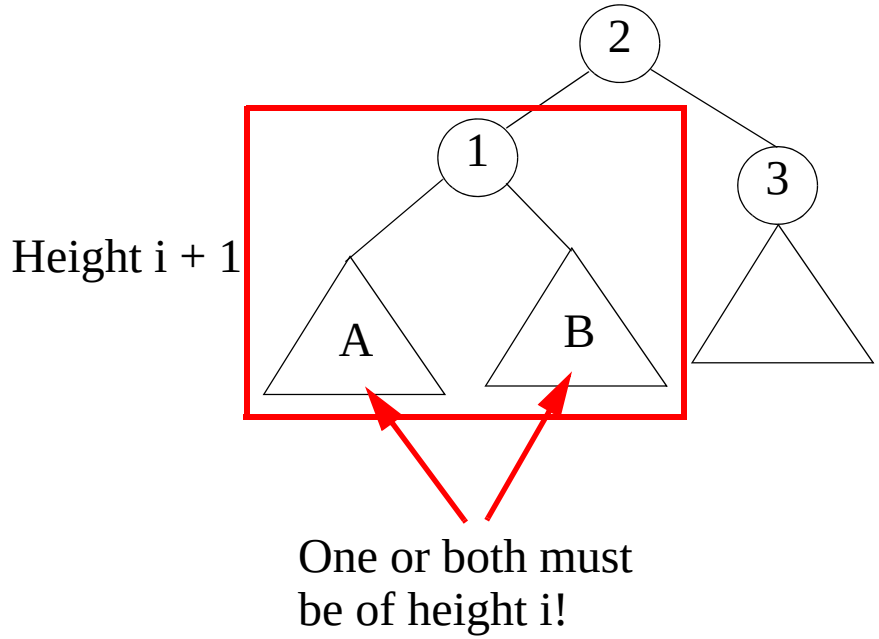
```
if (keyIn <= key)
  left = left.insert ();
else
  right = right.insert ();

if (left.getHeight () - right.getHeight () > 1)
  handleUnbalanceToLeft ();
else if (left.getHeight () - right.getHeight () < -1)
  handleUnbalanceToRight ();

setHeight ();
return this;
```

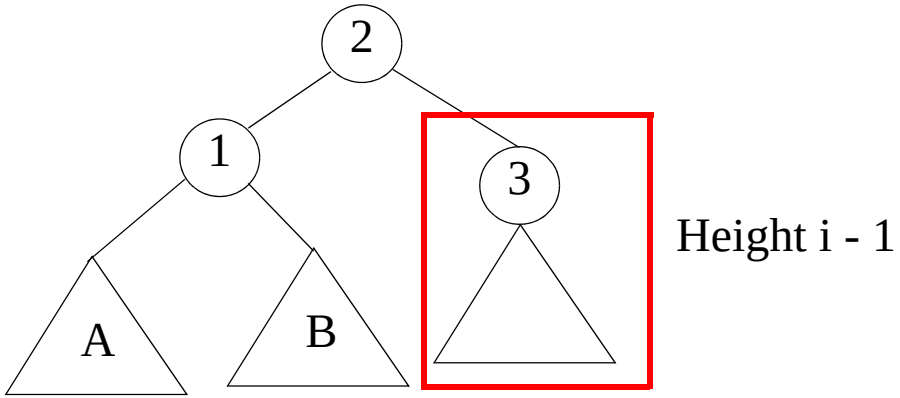
Handling Deletes

- In general, AVL deletes are quite complicated
- Our life is simpler because we only need to delete the **largest** key
  - — Less cases because we know we are always deleting from the **right** subtree
  - — When unbalanced after delete, picture is always like this:



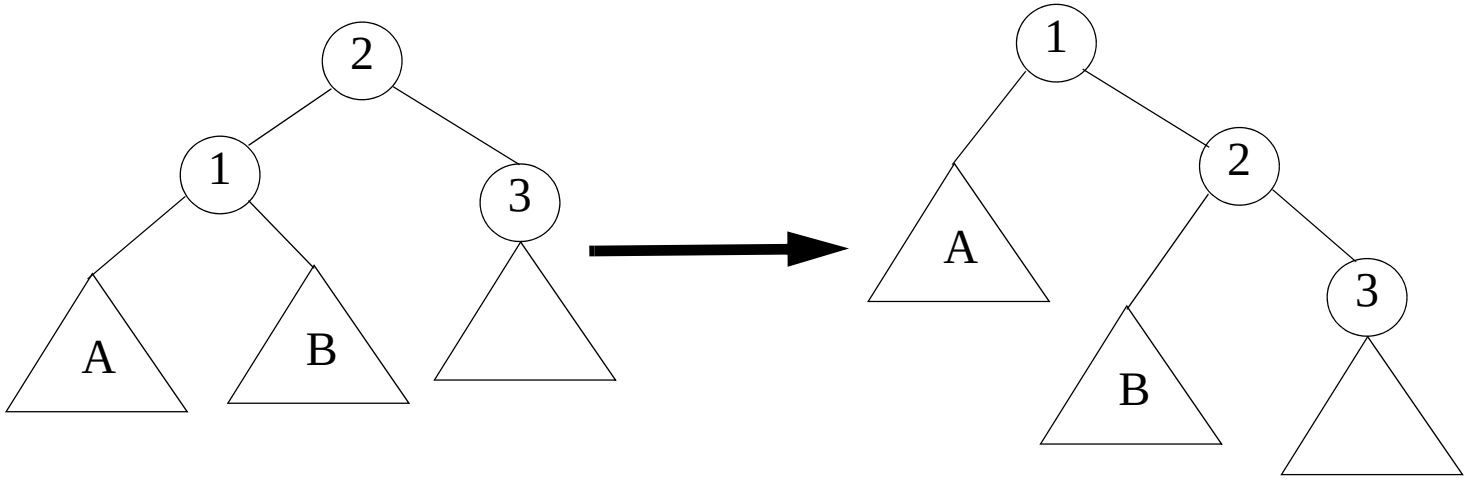
Handling Deletes

- In general, AVL deletes are quite complicated
- Our life is simpler because we only need to delete the **largest** key
  - — Less cases because we know we are always deleting from the **right** subtree
  - — When unbalanced after delete, picture is always like this:



Handling Deletes

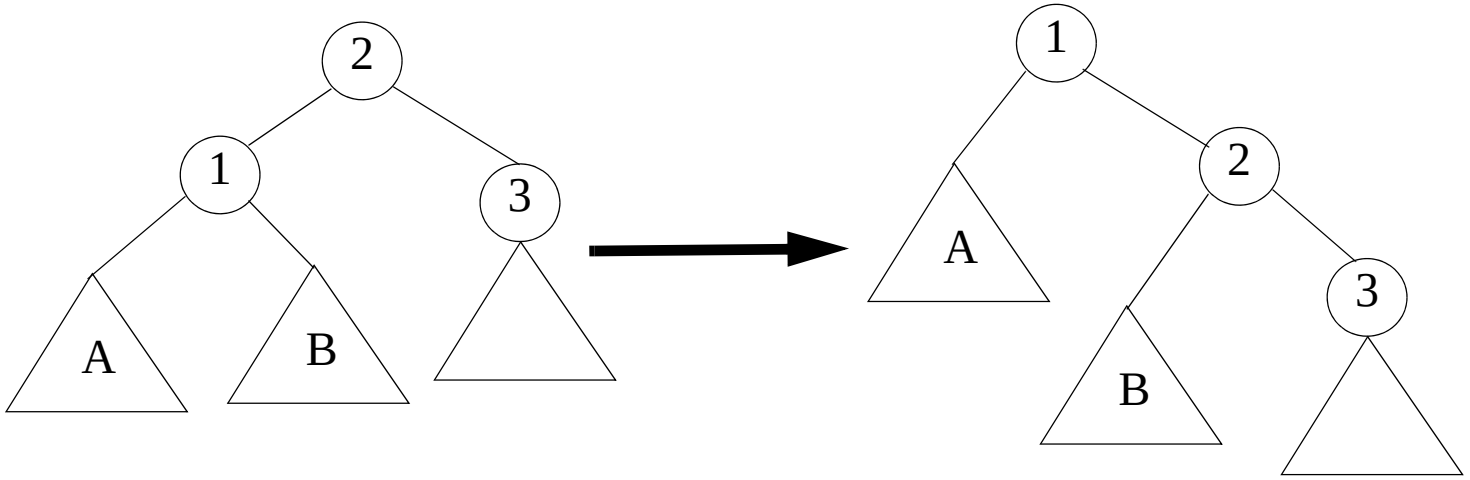
- Basic rotation to handle this is:



Caveat: won't always work!

Handling Deletes

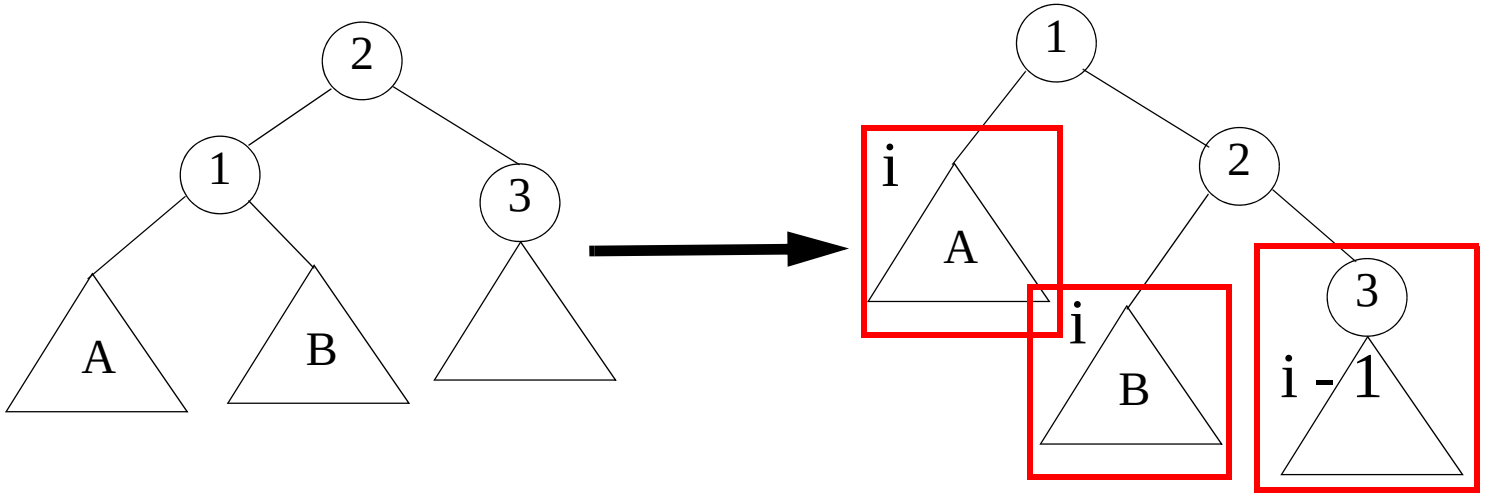
- Let us look at the various possible cases



Height of A	Height of B
i	i
i	i - 1
i - 1	i

Handling Deletes

- Let us look at the various possible cases

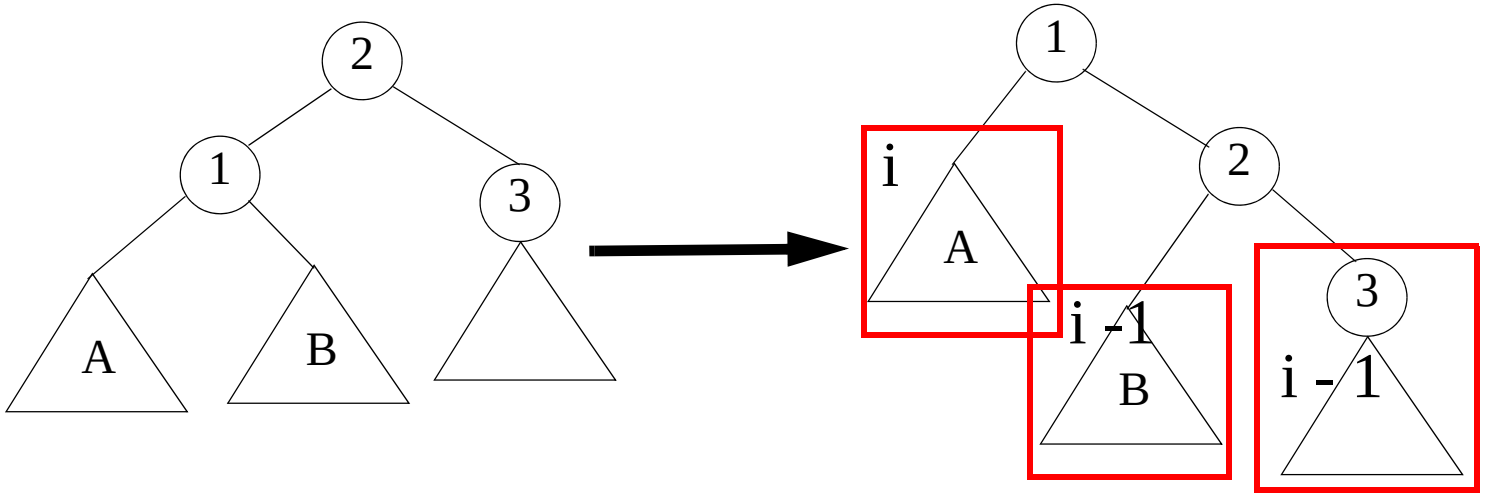


Height of A	Height of B
i	i
i	i - 1
i - 1	i

We are OK!

Handling Deletes

- Let us look at the various possible cases



Height of A

i	
i	
i - 1	

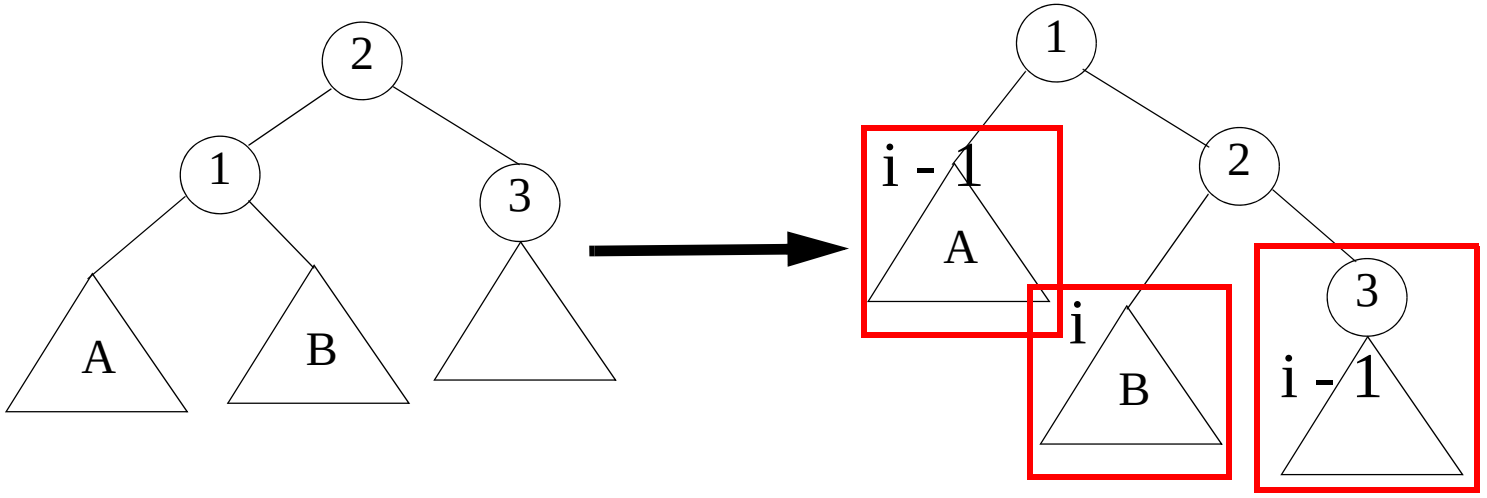
Height of B

i	
i - 1	
i	

We are OK here too!

Handling Deletes

- Let us look at the various possible cases

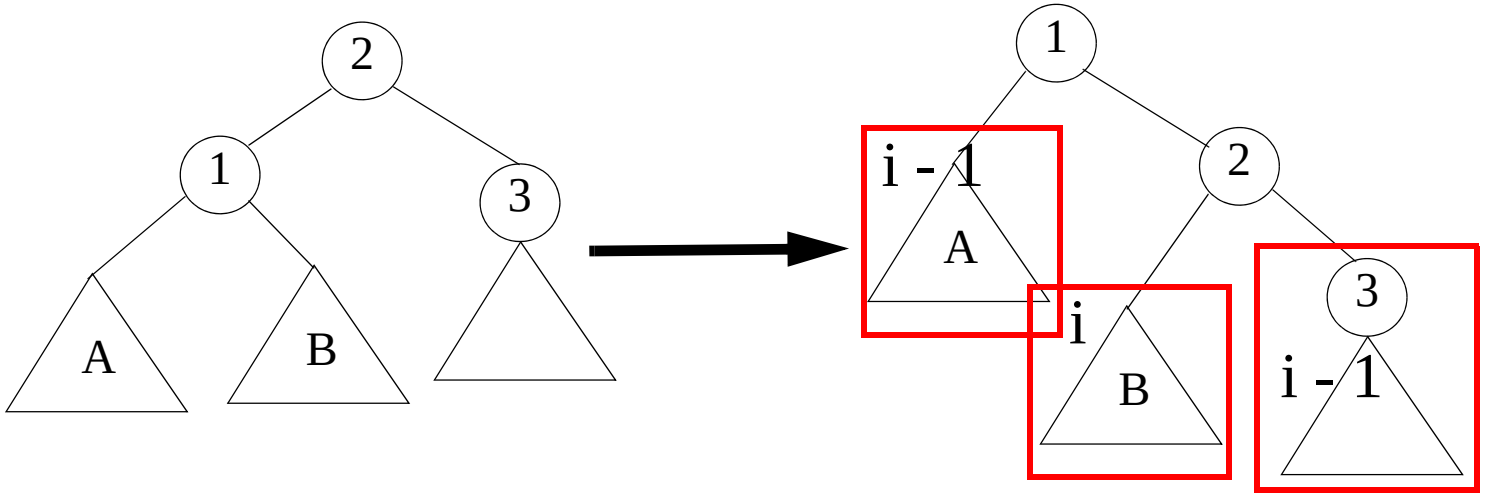


Height of A	Height of B
i	i
i	i - 1
i - 1	i

We are **not** OK here. Issue: tree rooted at 2 has depth of i + 1; tree A has depth of i - 1.

Handling Deletes

- Let us look at the various possible cases

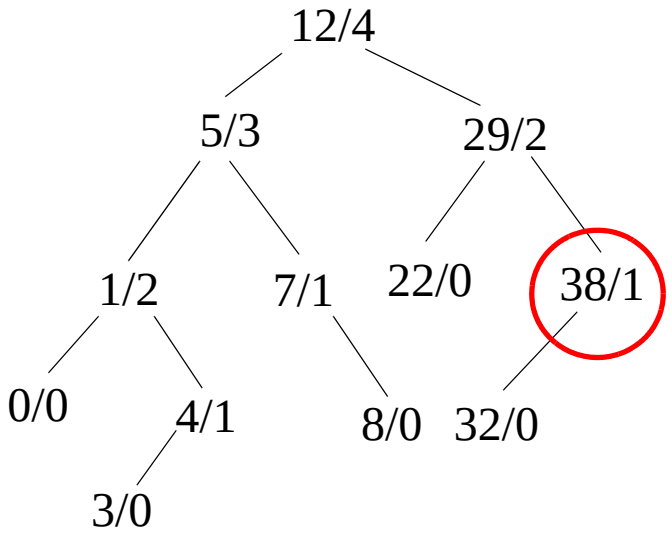


Height of A	Height of B
i	i
i	i - 1
i - 1	i

How to handle?
Not too bad!
Detect that tree rooted at 1 is right-deep, then call makeLeftDeep() on it, before you rotate.

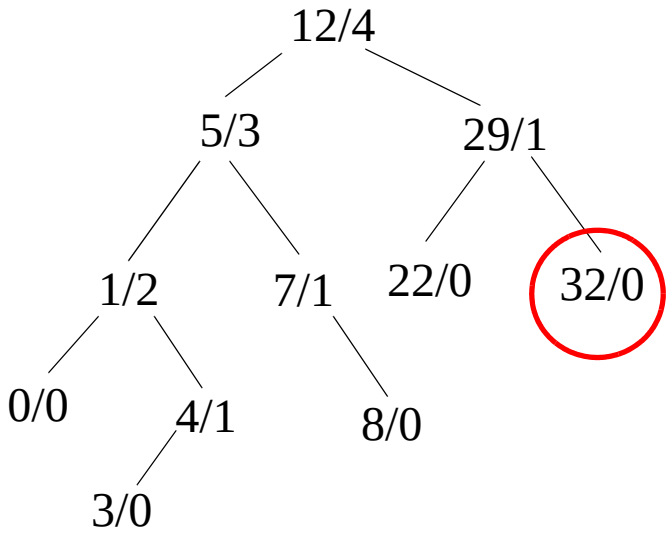
Handling Deletes

- In the end, removing the largest key looks a lot like last activity...
- Except that you always remove the key furthest to the right...



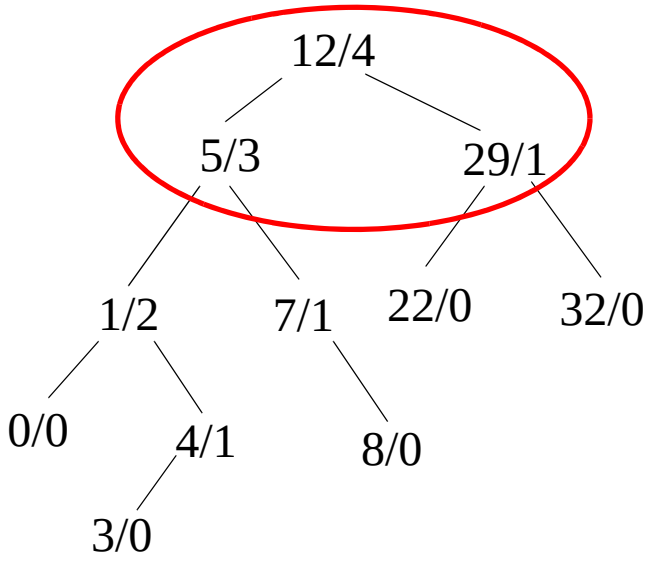
Handling Deletes

- In the end, removing the largest key looks a lot like last activity...
- Except that you always remove the key furthest to the right...
- And then move left subtree up to take its place



Handling Deletes

- In the end, removing the largest key looks a lot like last activity...
- Except that you always remove the key furthest to the right...
- And then move left subtree up to take its place
- Plus you have to detect imbalances after a recursive delete, and fix



Left and right differ by two...
So time to rotate!

Questions?