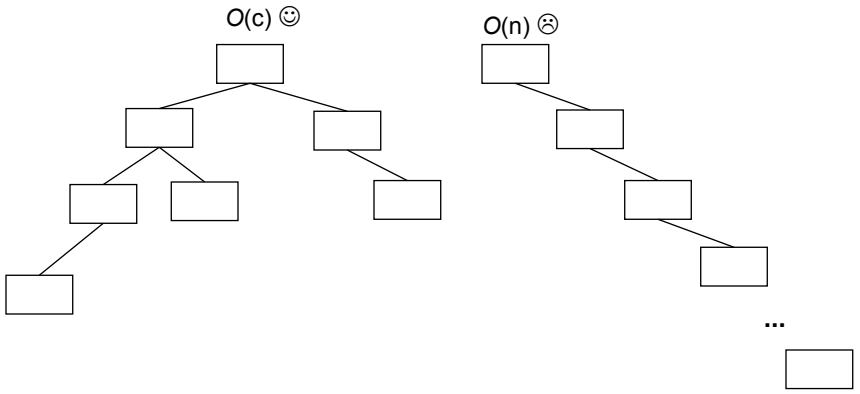
Balanced Trees

- In this section of notes you will learn about a special type of binary search tree: the balanced AVL tree

Recall: The Efficiency Of Most Tree Operations Is Dependent Upon The Height Of The Tree

Operation	Average case	Worse case
Search	$O(\log_2 n)$	$O(n)$
Insertion	$O(\log_2 n)$	$O(n)$
Deletion	$O(\log_2 n)$	$O(n)$

Balanced Trees Allow For Faster Operations Than Unbalanced Trees



A balanced tree: The height difference of the sub-trees of all nodes is either zero or one.

Balancing Of Binary Search Trees Is Highly Dependent Upon The Order Of The Inputs

- Contrast the trees built with the following data:
  - e.g. one, 20, 10, 30
  - e.g. two, 10 20 30

A Randomly Generated Binary Tree

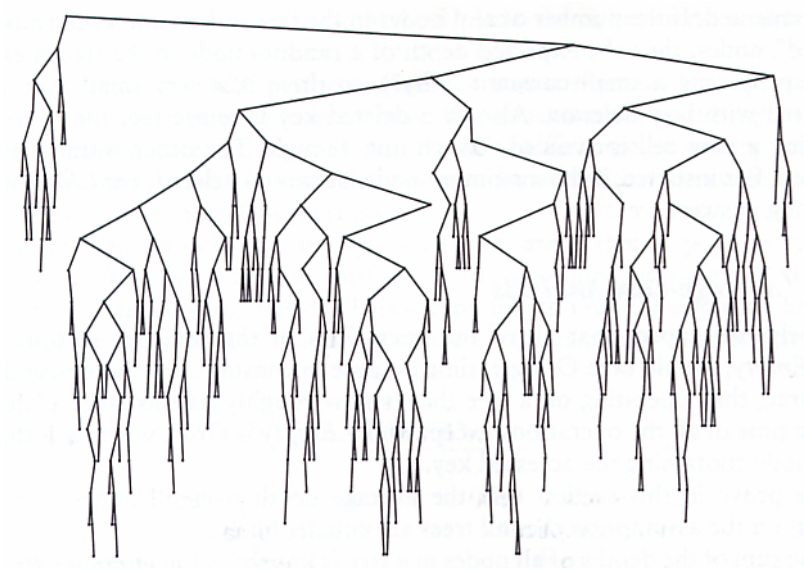


Image from “Data Structure and Algorithm Analysis in C++” by Mark Allen Weiss.

After Normal Use A Tree Can Quickly Become Unhinged

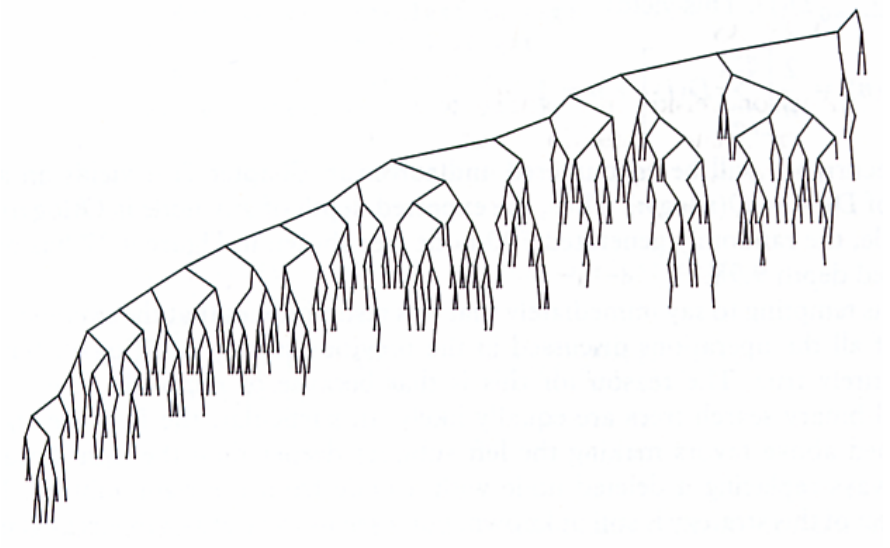


Image from “Data Structure and Algorithm Analysis in C++” by Mark Allen Weiss.

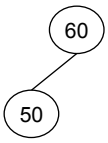
A Quick Comparison Of The Algorithms

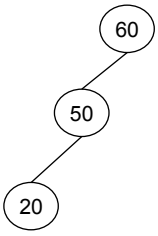
N	$T = Log_2 N$ (Height of Balanced tree)	T= root (N) (Height of tree after many operations)
1	0	1
2	1	1.41
:	:	:
32	5	5.66
64	6	8
1 million	20	1000

One Approach: Self Balancing Trees

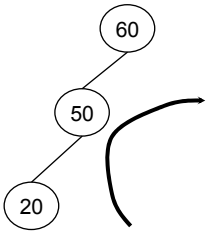
- As insertions and deletions are performed a check is made to determine if the tree is still balanced:
  - If the tree is still balanced then nothing more needs to be done
  - If the tree is now unbalanced then rearrange the nodes to re-balance the tree
- The type of self-balancing tree that will be covered are AVL trees (Adelson, Velskii & Landis)

Example Of Balancing A Tree

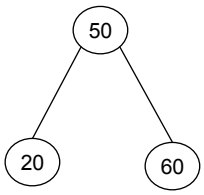




Problem: Tree is unbalanced ☹



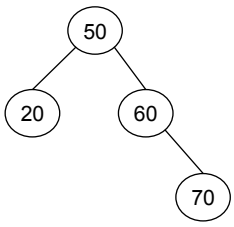
Solution: Re-balance tree by a right "rotation"

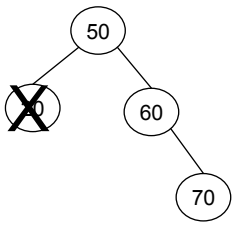


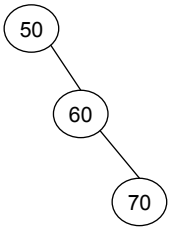
Tree is balanced again ☺

James Tam

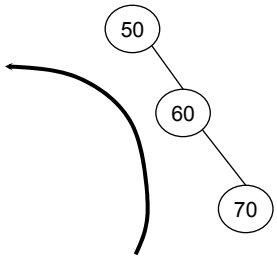
Example Of A Deletion Resulting In An Unbalanced Tree



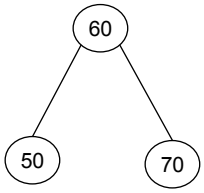




Problem: Tree is unbalanced ☹



Solution: Re-balance tree by a left "rotation"



Tree is balanced again ☺

James Tam

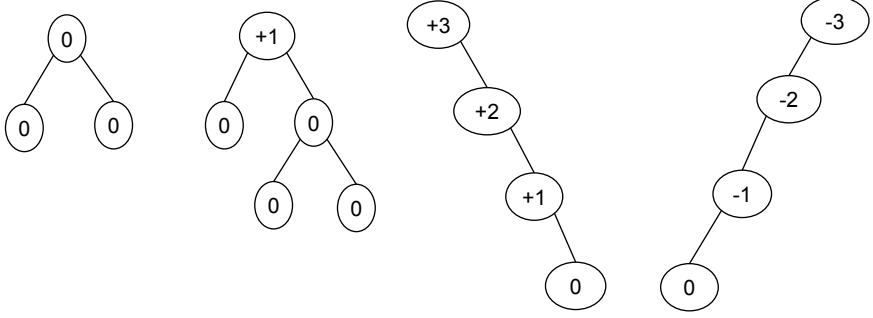
Balance Factors

- In order to determine if the height of the left and right sub-trees for a particular node are balanced you need add an additional attribute to the Node class.
- The balance factor is equal to the difference between the height of the right and left sub-trees of that node.

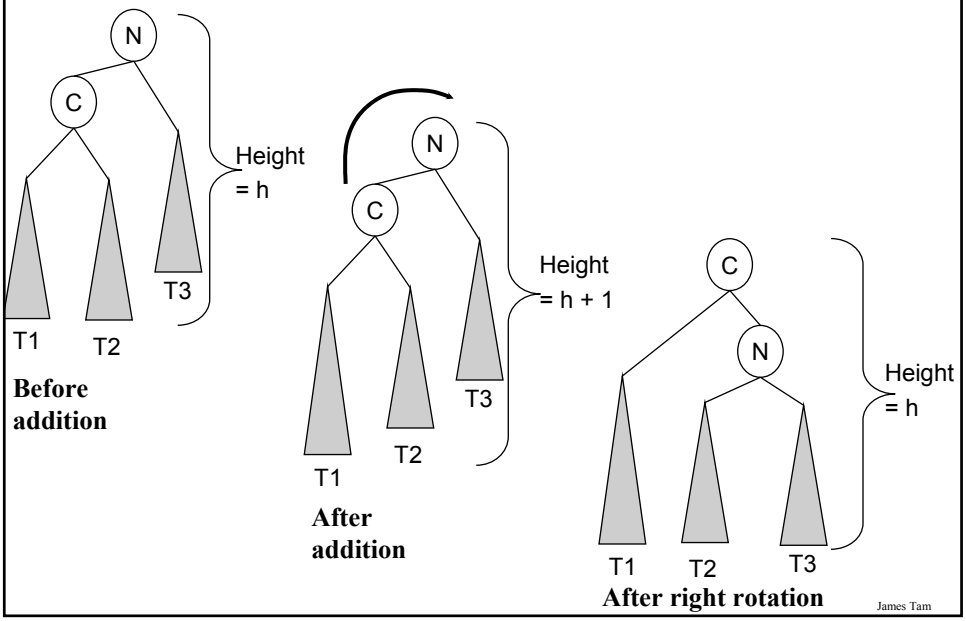
```
public class AVLNode extends BinaryNode
{
    private int balanceFactor;
    :      :      :
}
```


Balance Factors(2)

- Examples (right - left)



Graphically Illustrating The Single Right Rotation



Algorithm For Performing Single Right Rotations

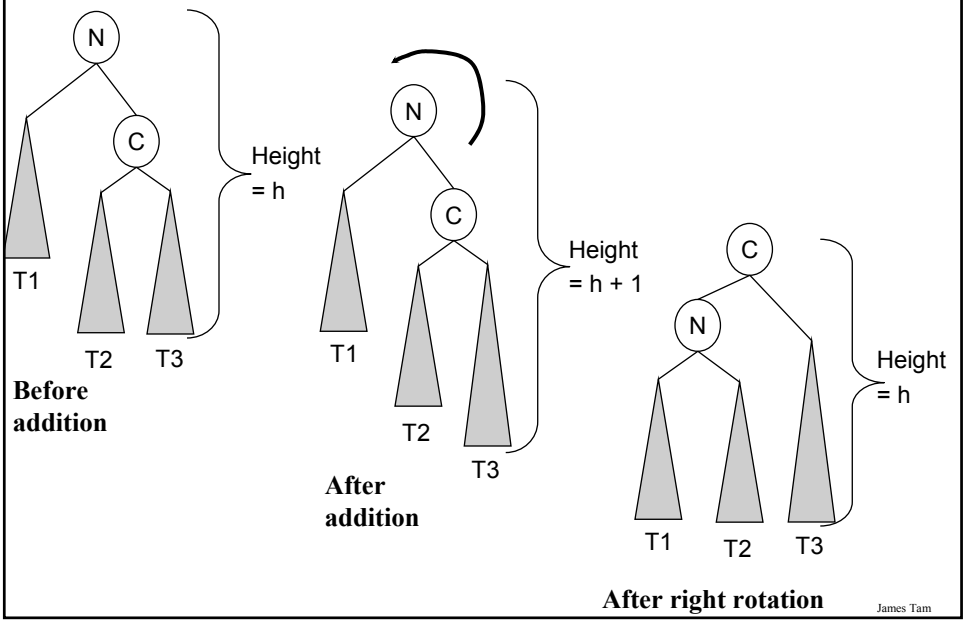
- Prior to the addition or deletion the tree was balanced.
- The addition or deletion of a node causes an imbalance
- Starting at the newly inserted node and working towards the root find the first unbalanced node "N". Balance that node and you are done.[1]
  - If node "C" is the left child of node N:
  - Set node N's left child to node C's right child
  - Set node C's right child to refer to node N

Pseudo-Code:

```
rotateRight (Node n)
{
    Node c = n.getLeft ()
    n.setLeft ( c.getRight() )
    c.setRight ( n )
}
```

1 Obviously for all rotations Node C must take the place of Node N in the tree: a) if Node N was the root then Node C becomes the new root. b) otherwise Node N's parent will refer to Node C instead of Node N.

Graphically Illustrating The Single Left Rotation



Algorithm For Performing Single Left Rotations

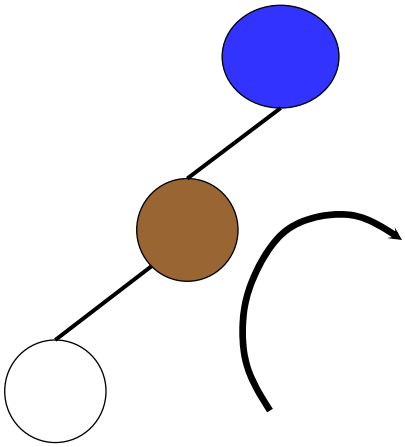
- Prior to the addition or deletion the tree was balanced.
- The addition or deletion of a node causes an imbalance
- Starting at the newly inserted node and working towards the root find the first unbalanced node "N". Balance that node and you are done.
  - If node "C" is the right child of node N:
  - Set node N's right child to node C's left child
  - Set node C's left child to refer to node N

Pseudo-Code:

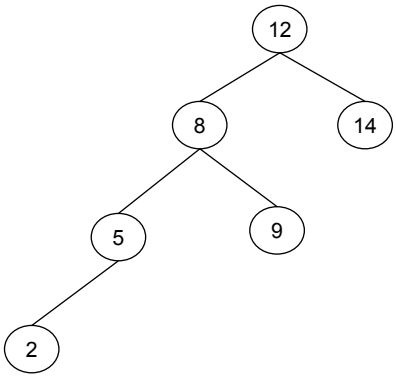
```
rotateLeft: (Node n)
{
    Node c = n.getRight ()
    n.setRight (c.getLeft() )
    c.setLeft ( n )
}
```

Spotting Single Rotations: Left-Left Imbalance

- Balance factor of the parent (unbalanced node) = -2
- Balance factor the left child = -1

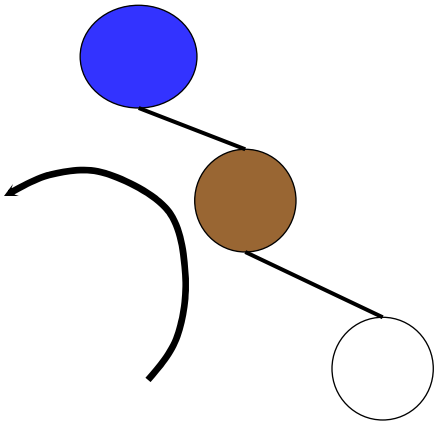


Example: Single Right Rotation For Left-Left Imbalance

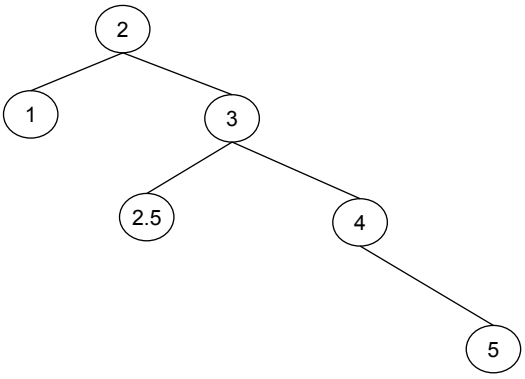


Spotting Single Rotations: Right-Right Imbalance

- Balance factor of parent (unbalanced node) = +2
- Balance factor of the right child = + 1

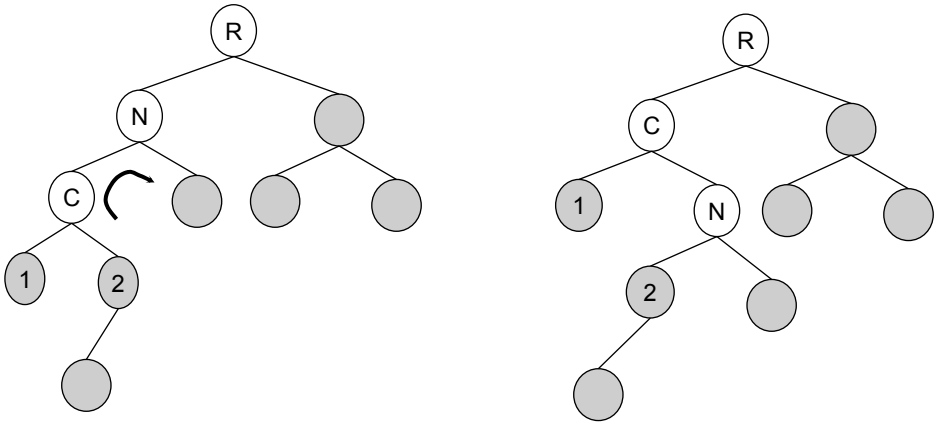


Example: Single Left Rotation For Right-Right Imbalance



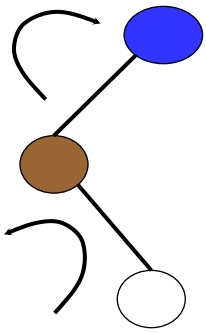
Spotting Double Rotations: Left-Right Imbalance

- A single rotation will not fix the problem
- Single rotation: Rotating node n right won't work



Spotting Double Rotations: Left-Right Imbalance (2)

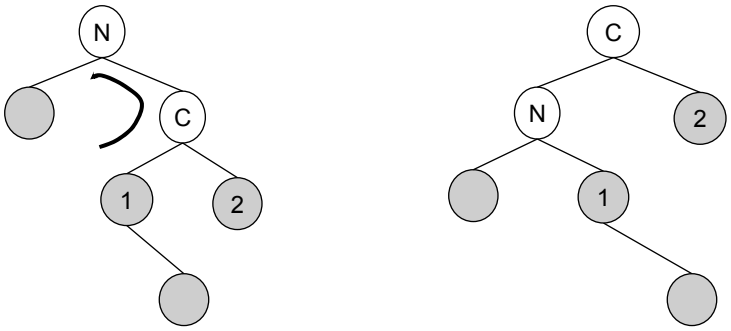
- Balance factor of parent (unbalanced node) = - 2
- Balance factor of child = +1



Spotting Double Rotations: Right-Left Imbalance

•A single rotation will not fix the problem.

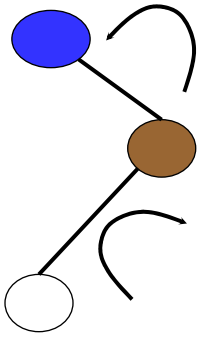
•Single rotation: Rotating node n left won't work



Spotting Double Rotations: Right-Left Imbalance (2)

•Balance factor of the parent (unbalanced node): +2

•Balance factor of the child: -1



Algorithm For Performing Left-Right Rotations

- An imbalance occurs for node N if an insertion occurs in the left sub-tree of node N's right sub child
- Given that node "C" is the left child of N.
  - Rotate left around child "C"
  - Rotate right around the parent "N"

Pseudo-Code:

```
rotateLeftRight: (Node n)
{
    Node c = n.getLeft ()
    n.setLeft ( rotateLeft ( c)  )
    rotateRight ( n )
}
```


Graphically Illustrating Left-Right Rotations

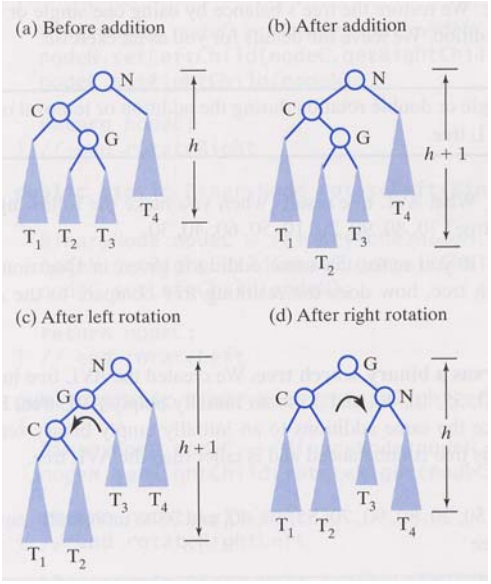


Image from *"Data Structures and Abstractions with Java"* by Frank M. Carrano and Walter Savitch

Algorithm For Performing Right-Left Rotations

- An imbalance occurs for node N if an insertion occurs in the right sub-tree of node N's left sub child
- Given that node "C" is the right child of N.
  - Rotate right around the child "C"
  - Rotate left around the parent "P"

Pseudo-Code:

```
rotateLeftRight: (Node n)
{
    Node c = n.getRight ()
    n.setRight ( rotateRight ( c)  )
    rotateLeft ( n )
}
```


Graphically Illustrating Right-Left Rotations

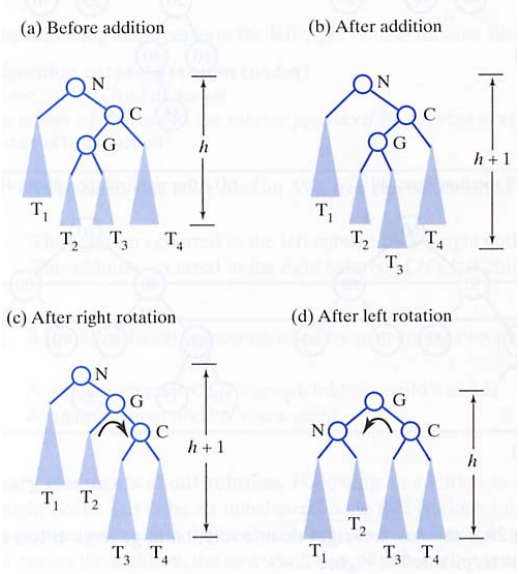


Image from *"Data Structures and Abstractions with Java"* by Frank M. Carrano and Walter Savitch

Summary Table Of Rotations

Type of imbalance		Balance factor of parent	Balance factor of child	Direction of 1st rotation	Direction of 2nd Rotation
Left-left		-2	-1	Right	NA
Right-right		+2	+1	Left	NA
Left-right		-2	+1	Left (child)	Right (node)
Right-left		+2	-1	Right (child)	Left (node)

Efficiency Of AVL Trees

- Effect of balancing on other tree operations
  - Typically the height is ~ 1.44 *($\log_2 N$) which effects searches, insertions and deletions
- Efficiency of the balance operation itself
  - Search time (to find the imbalance): $\log_2 N$
  - Time to perform the rotation: Constant time

You Should Now Know

- How additions and deletions to a tree can cause it become unbalanced?
- How balance factors can be used to determine if a node has an unbalanced left and right sub tree
- How to determine the type of imbalance
  - Left-left
  - Right-right
  - Left-right
  - Right-left
- How to rebalance the different types of unbalanced trees

Sources Of Lecture Material

- "*Data Abstraction and Problem Solving with Java: Walls and Mirrors*" by Frank M. Carrano and Janet J. Prichard
- *"Data Structures and Abstractions with Java"* by Frank M. Carrano and Walter Savitch
- "Data Structures and Algorithms in Java" by Adam Drozdek
- The Wiley Science web site http://www3.interscience.wiley.com:8100/legacy/college/koffman/0471467561/ppt/ch11.ppt
- CPSC 331 course notes by Marina L. Gavrilova http://pages.cpsc.ucalgary.ca/~marina/331/