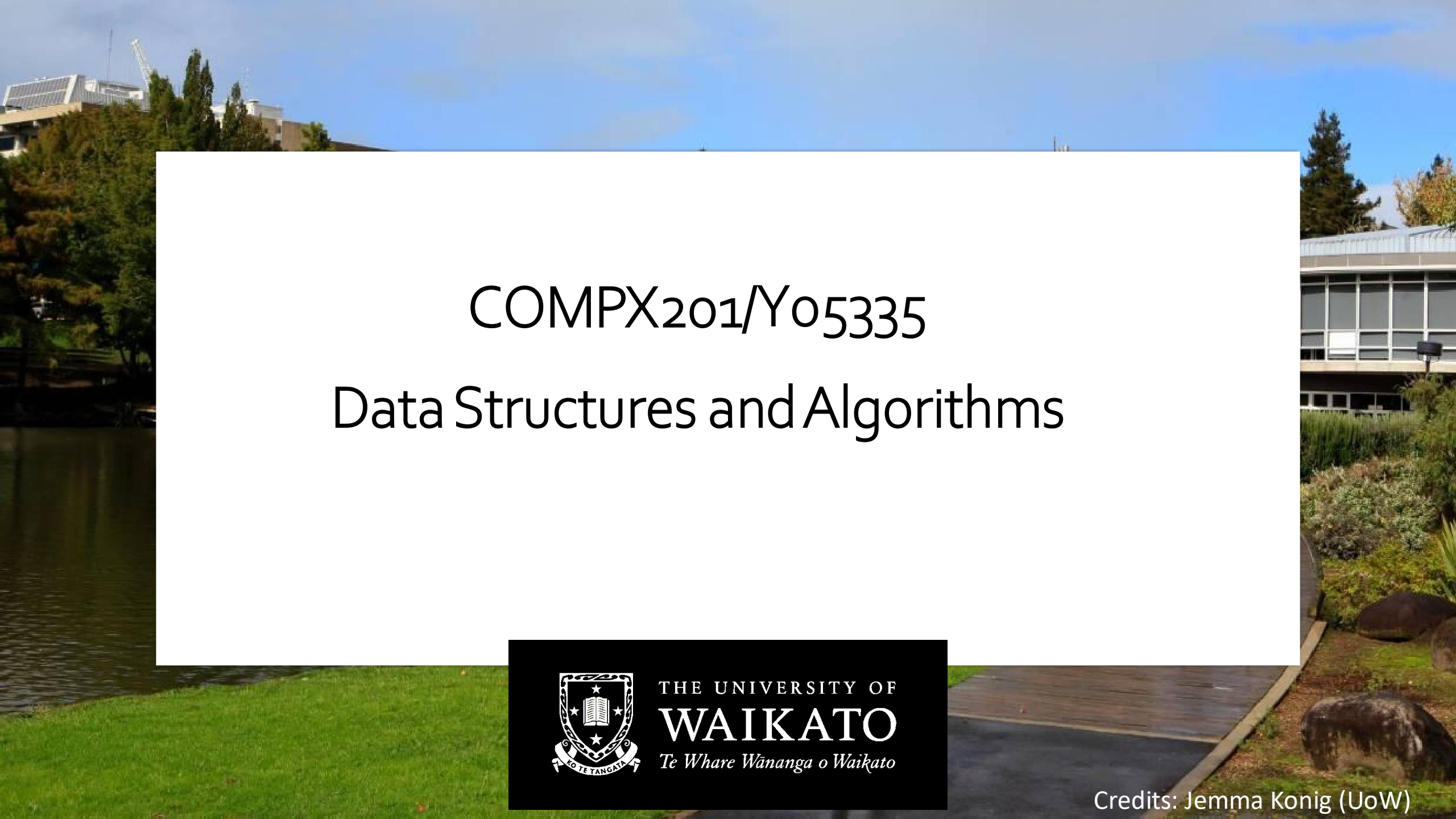


COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

AVL trees in Java
Height and BF

COMPX201/Y05335

Overview

- AVL operations
- Height
- Balance Factor

Operations

AVL Tree: Operations

- Search
- Insert
- Delete
- Traversal
- Height
- Balance Factor
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

AVL Tree: Operations

- *Search - same as BST*
- Insert
- Delete
- *Traversal - same as BST*
- Height
- Balance Factor
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

AVL Tree: Operations

- Insert
- Delete
- Height
- Balance Factor
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

AVL Tree: Operations

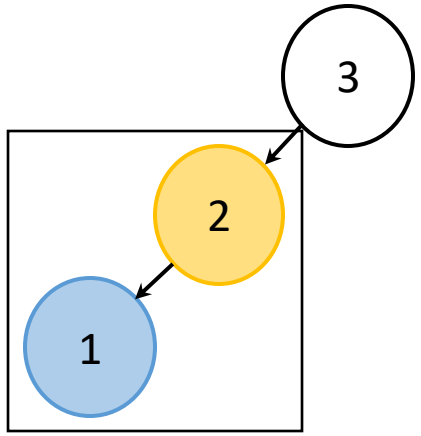
- Insert
- Delete
- Height
- Balance Factor
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

Height

Height

- Height is the longest path from some node to the leaf

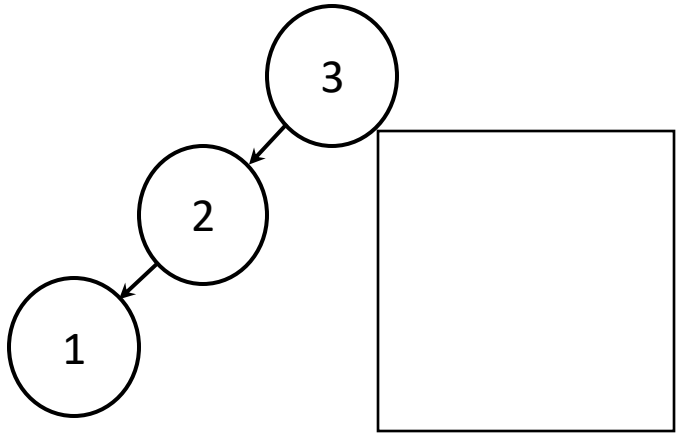
Height = 1



Height

- Height is the longest path from some node to the leaf
- Remember, an empty sub-tree has a height of -1

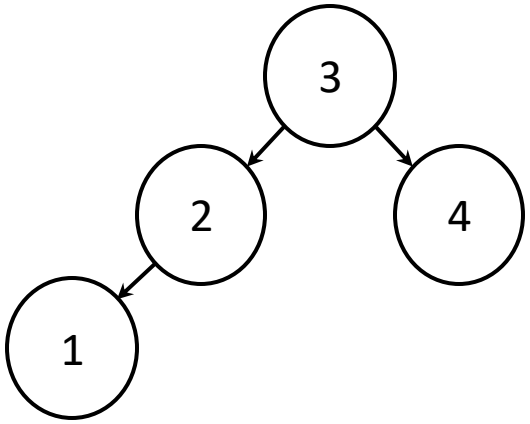
Height = 1



Height = -1

Height

- So, we can say that the height of a node is:
  - -1 if the node is null
  - Or, if node is not null, then
  - Height is the greater height between the left and right subtrees, plus 1



Height

- So, we can say that the height of a node is:
  - -1 if the node is null
  - Or, if node is not null, then
  - Height is the greater height between the left and right subtrees, plus 1

```
IF node is null
    RETURN -1
ELSE
    GET height of left subtree
    GET height of right subtree
    CALCULATE max of two heights
    ADD one
    RETURN answer
```

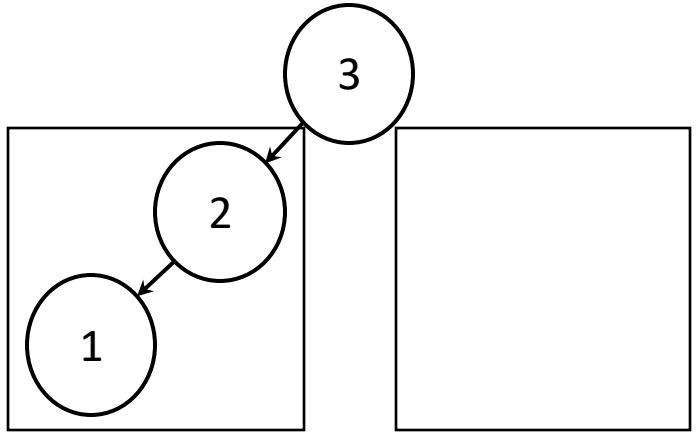
Height

- Okay, lets try it …

Balance factor

Balance factor

LST height = 1



RST height = -1

BF = 1 − − 1

BF = 1 + 1

BF = 2

Balance factor

- So, we can say that the balance factor of a node is:
  - 0 if the node is null
  - Height of the left subtree – height of the right subtree

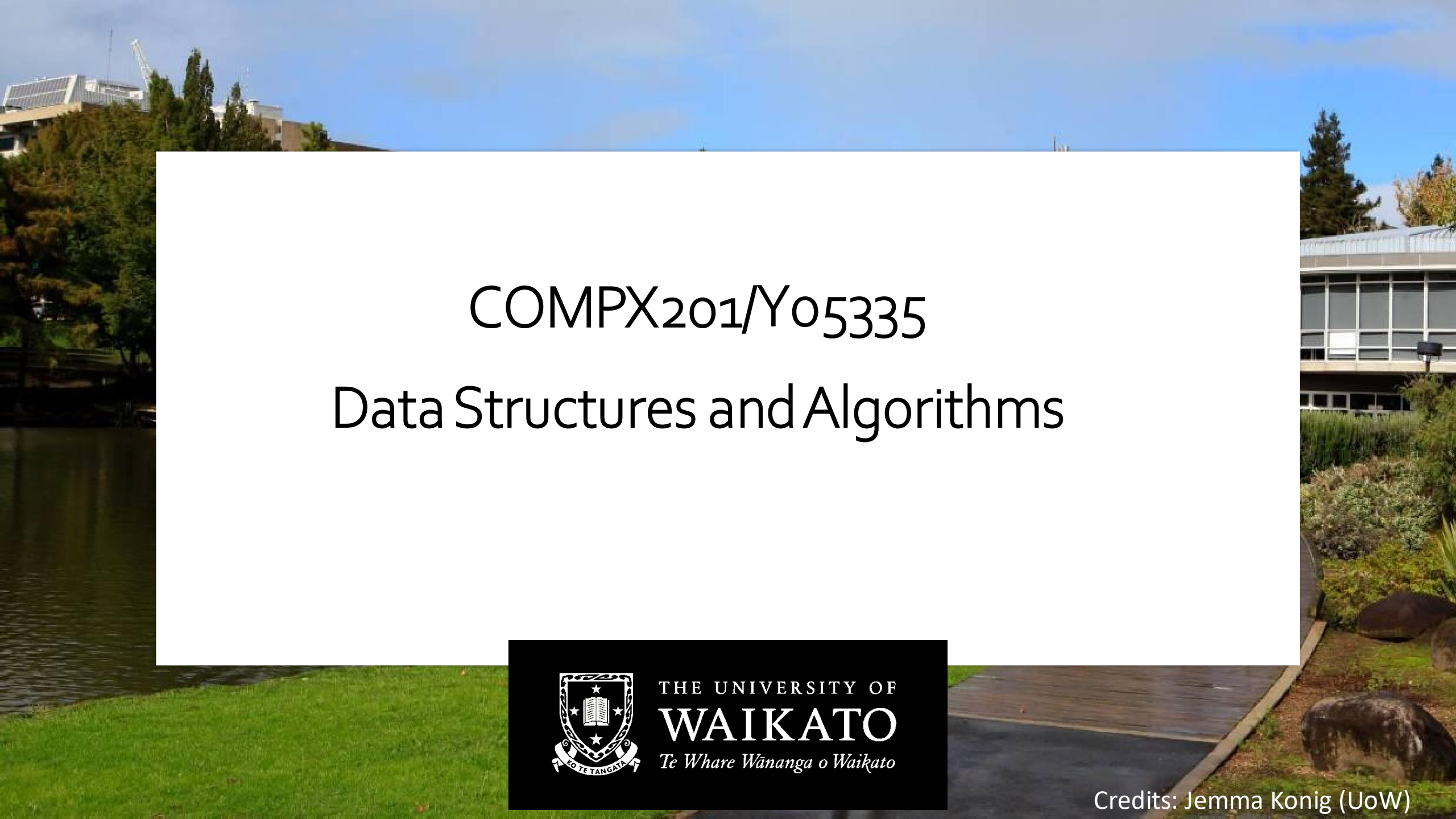
Balance factor

- So, we can say that the balance factor of a node is:
  - 0 if the node is null
  - Height of the left subtree – height of the right subtree

```
IF node is null
    RETURN 0
ELSE
    CALCULATE height LST - height RST
    RETURN answer
```

Balance factor

- Okay, lets try it …



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

AVL trees in Java
Rotations

COMPX201/Y05335

Overview

- AVL operations
- Left rotation
- Right rotation
- Left-right rotation
- Right-left rotation

Operations

AVL Tree: Operations

- Insert
- Delete
- Height
- Balance Factor
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

AVL Tree: Operations

- Insert
- Delete
- ~~Height~~
- ~~Balance Factor~~
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

AVL Tree: Operations

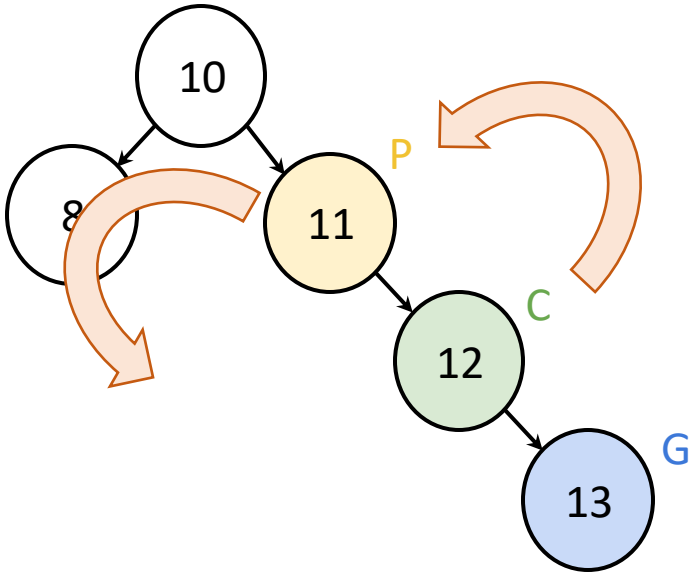
- Insert
- Delete
- ~~Height~~
- ~~Balance Factor~~
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

Left rotation

AVL Tree: Left rotation

- If extra height exists down the right-subtree of a right-subtree
- Then the tree is right-imbalanced, so ROTATE LEFT.
- Rotate left:
  - Point parent.right to child.left
  - Point child.left to parent
  - Return child

P = parent
C = child
G = grandchild



Left rotation

- So, we can rotate a node left by:
  - Passing in the parent
  - Getting access to the right child
  - Pointing parent.right to child.left
  - Pointing child.left to parent
  - Returning child

Left rotation

- So, we can rotate a node left by:
  - Passing in the parent
  - Getting access to the right child
  - Pointing parent.right to child.left
  - Pointing child.left to parent
  - Returning child

```
DEFINE child as parent.right
SET parent.right to be child.left
SET child.left to be parent
RETURN child
```

Left rotation

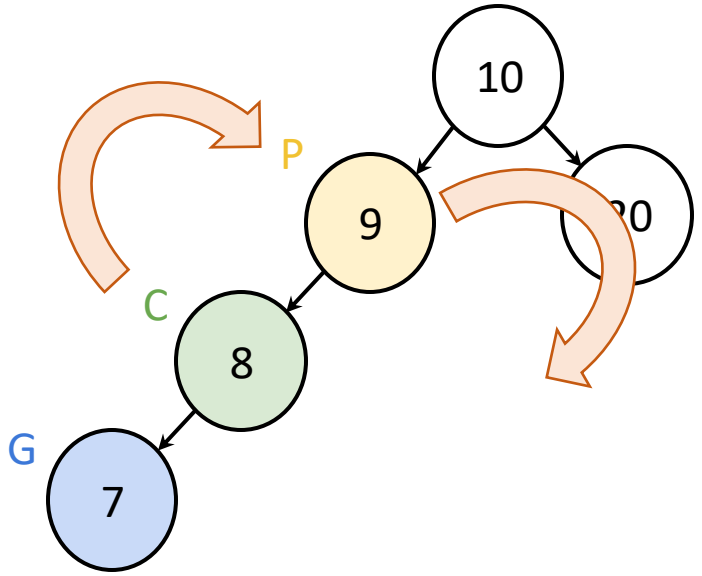
- Okay, lets try it …

Right rotation

AVL Tree: Right rotation

- If extra height exists down the left-subtree of a left-subtree
- Then the tree is left-imbalanced, so ROTATE RIGHT.
- Rotate right:
  - Point parent.left to child.right
  - Point child.right to parent
  - Return child

P = parent
C = child
G = grandchild



Right rotation

- So, we can rotate a node right by:
  - Passing in the parent
  - Getting access to the left child
  - Pointing parent.left to child.right
  - Pointing child.right to parent
  - Returning child

Right rotation

- So, we can rotate a node right by:
  - Passing in the parent
  - Getting access to the left child
  - Pointing parent.left to child.right
  - Pointing child.right to parent
  - Returning child

```
DEFINE child as parent.left
SET parent.left to be child.right
SET child.right to be parent
RETURN child
```

Right rotation

- Okay, lets try it …

Left-right rotation

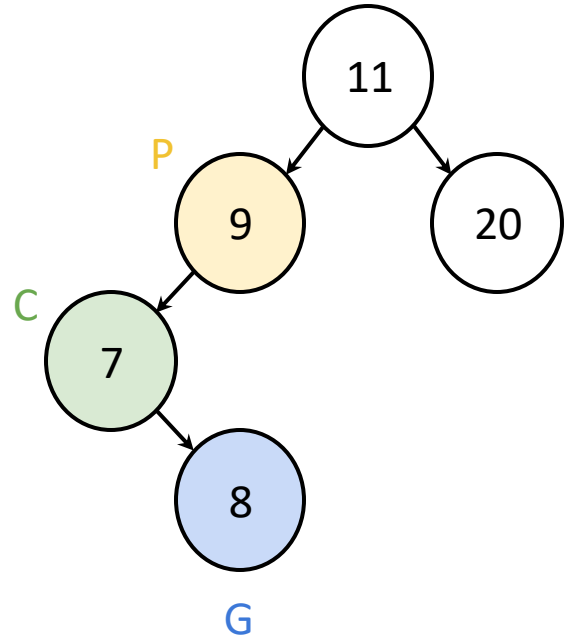
AVL Tree: Left-right rotation

- If extra height exists down the right-subtree of a left-subtree

- Then the tree is left-right imbalanced, so ROTATE LEFT-RIGHT.

- First, rotate left (on parent.left):
  - Point child.right to grandchild.left
  - Point grandchild.left to child
  - Return grandchild
- Then, rotate right (on parent):
  - Point parent.left to child.right
  - Point child.right to parent
  - Return child

P = parent
C = child
G = grandchild



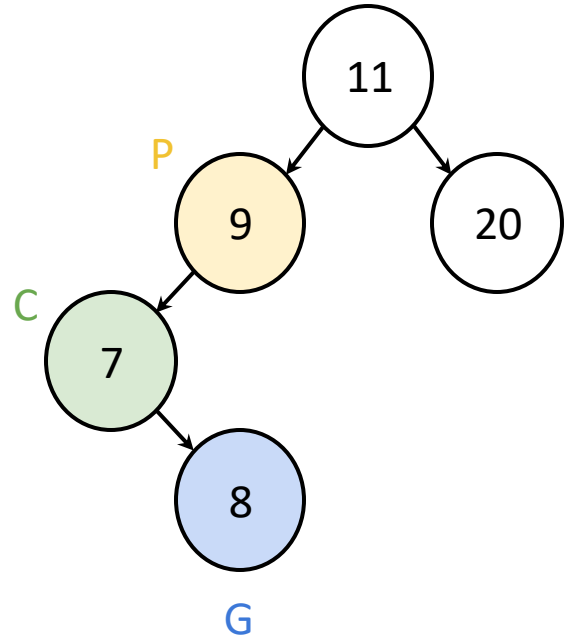
AVL Tree: Left-right rotation

- If extra height exists down the right-subtree of a left-subtree

- Then the tree is left-right imbalanced, so ROTATE LEFT-RIGHT.

- First, rotate left (on parent.left):
  - Point child.right to grandchild.left
  - Point grandchild.left to child
  - Return grandchild
- Then, rotate right (on parent):
  - Point parent.left to child.right
  - Point child.right to parent
  - Return child

P = parent
C = child
G = grandchild



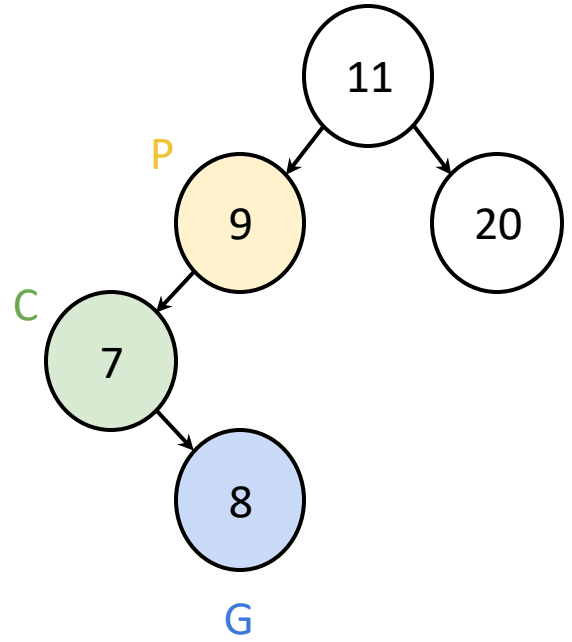
AVL Tree: Left-right rotation

- If extra height exists down the right-subtree of a left-subtree

- Then the tree is left-right imbalanced, so ROTATE LEFT-RIGHT.

- First, rotate left (on parent.left):
  - Point child.right to grandchild.left
  - Point grandchild.left to child
  - Return grandchild
- Then, rotate right (on parent):
  - Point parent.left to child.right
  - Point child.right to parent
  - Return child

P = parent
C = child
G = grandchild



Left-right rotation

- So, we can rotate a node left-right by:
  - Rotating left – on parent.left
  - Rotating right – on parent

Left-right rotation

- So, we can rotate a node left-right by:
  - Rotating left – on parent.left
  - Rotating right – on parent

```
cRoot.left = rotateLeft(cRoot.left);
return rotateRight(cRoot);
```

Left-right rotation

Okay, lets try it ...

Right-left rotation

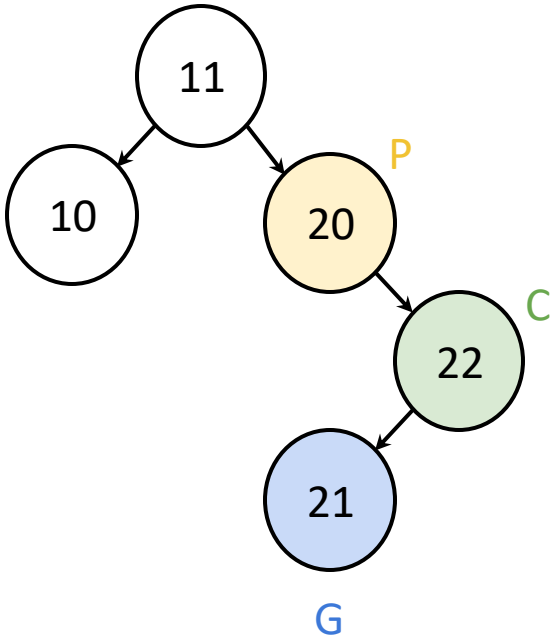
AVL Tree: Right-left rotation

- If extra height exists down the left-subtree of a right-subtree

- Then the tree is right-left imbalanced, so ROTATE RIGHT-LEFT.

- First, rotate right (on parent.right):
  - Point child.left to grandchild.right
  - Point grandchild.right to child
  - Return grandchild
- Then, rotate left (on parent):
  - Point parent.right to child.left
  - Point child.left to parent
  - Return child

P = parent
C = child
G = grandchild



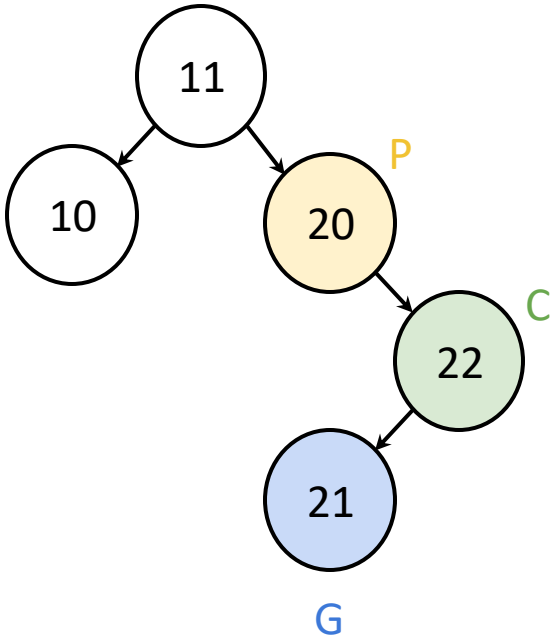
AVL Tree: Right-left rotation

- If extra height exists down the left-subtree of a right-subtree

- Then the tree is right-left imbalanced, so ROTATE RIGHT-LEFT.

- First, rotate right (on parent.right):
  - Point child.left to grandchild.right
  - Point grandchild.right to child
  - Return grandchild
- Then, rotate left (on parent):
  - Point parent.right to child.left
  - Point child.left to parent
  - Return child

P = parent
C = child
G = grandchild



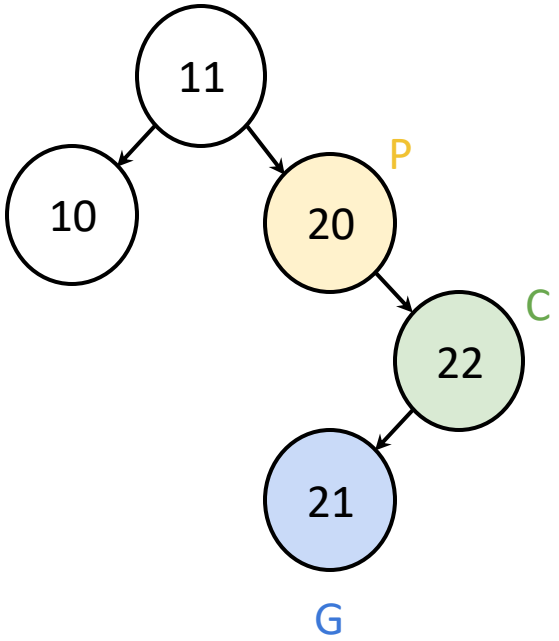
AVL Tree: Right-left rotation

- If extra height exists down the left-subtree of a right-subtree

- Then the tree is right-left imbalanced, so ROTATE RIGHT-LEFT.

- First, rotate right (on parent.right):
  - Point child.left to grandchild.right
  - Point grandchild.right to child
  - Return grandchild
- Then, rotate left (on parent):
  - Point parent.right to child.left
  - Point child.left to parent
  - Return child

P = parent
C = child
G = grandchild



Right-left rotation

- So, we can rotate a node right-left by:
  - Rotating right – on parent.right
  - Rotating left – on parent

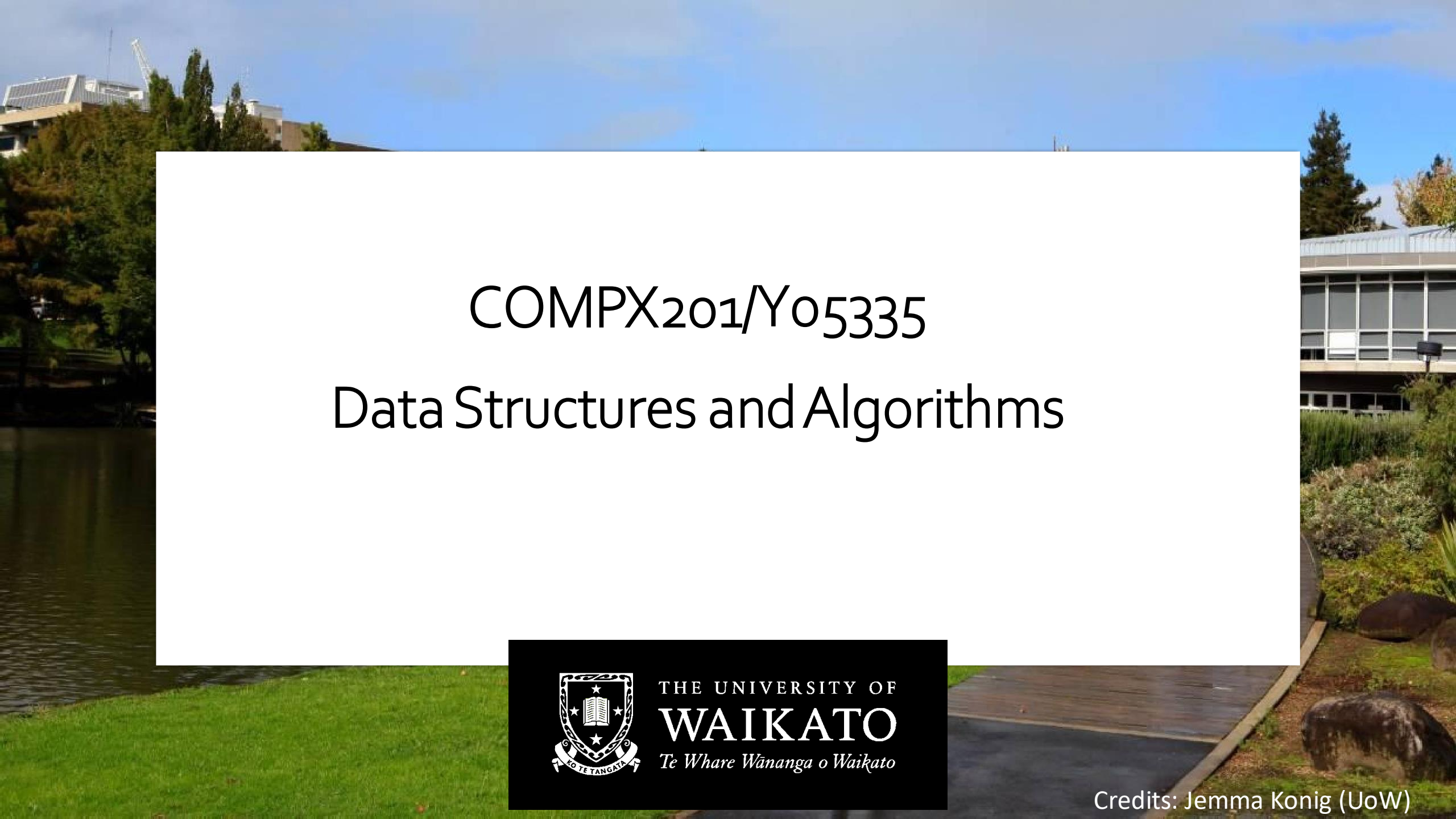
Right-left rotation

- So, we can rotate a node right-left by:
  - Rotating right – on parent.right
  - Rotating left – on parent

```
cRoot.right = rotateRight(cRoot.right);
return rotateLeft(cRoot);
```

Right-left rotation

Okay, lets try it …



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

AVL trees in Java
Insertion

COMPX201/Y05335

Overview

- AVL operations
- Insert

Operations

AVL Tree: Operations

- Insert
- Delete
- Height
- Balance Factor
- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

AVL Tree: Operations

- Insert
- Delete
- ~~Height~~
- ~~Balance Factor~~
- ~~Left rotation~~
- ~~Right rotation~~
- ~~Left-right rotation~~
- ~~Right-Left rotation~~

AVL Tree: Operations

- Insert
- Delete
- ~~Height~~
- ~~Balance Factor~~
- ~~Left rotation~~
- ~~Right rotation~~
- ~~Left-right rotation~~
- ~~Right-Left rotation~~

AVL Tree: Operations

- Insert
- Delete – *I'm going to leave delete for you to try!*
- ~~Height~~
- ~~Balance Factor~~
- ~~Left rotation~~
- ~~Right rotation~~
- ~~Left-right rotation~~
- ~~Right-Left rotation~~

Insert

Insert

- … we already know how to insert into a BST

BST insertion

- If the root is empty, use new node as root.
- Otherwise we need to compare and determine which subtree the node belongs in.
- If less than current root, put in left subtree.
- If greater than current root, put in right subtree.

BST insertion

```
public void insert (int x){
        // Call recursive insertion method
        root = insertR(root, 12);
}
```

```
public Node insertR(Node cRoot, int x){
        // No tree so add here
        if (cRoot == null){
                cRoot = new Node(x);
        }
        // Value is smaller
        else if (x < cRoot.value){
                cRoot.left = insertR(cRoot.left, x);
        }
        // Value is larger
        else if (x > cRoot.value){
                cRoot.right = insertR(cRoot.right, x);

        }
        return cRoot;
}
```

Insert

- … we already know how to insert into a BST

- For an AVL tree, we need to insert, check for imbalance, then balance tree
  - Insert node
  - Calculate balance factor
  - If balance factor is greater than 1 or less than -1,
  - Rotate to correct for imbalance

Insert

- … we already know how to insert into a BST

- For an AVL tree, we need to insert, check for imbalance, then balance tree
  - Insert node
  - Calculate balance factor
  - If balance factor is greater than 1 or less than -1,
  - Rotate to correct for imbalance

- BUT how do we know which way to rotate?

Insert

- Lets start with:
  - Insert node
  - Calculate balance factor
  - If balance factor is greater than 1 or less than -1,
  - Then print “this tree is imbalanced!”

Insert

- Okay, lets try it …

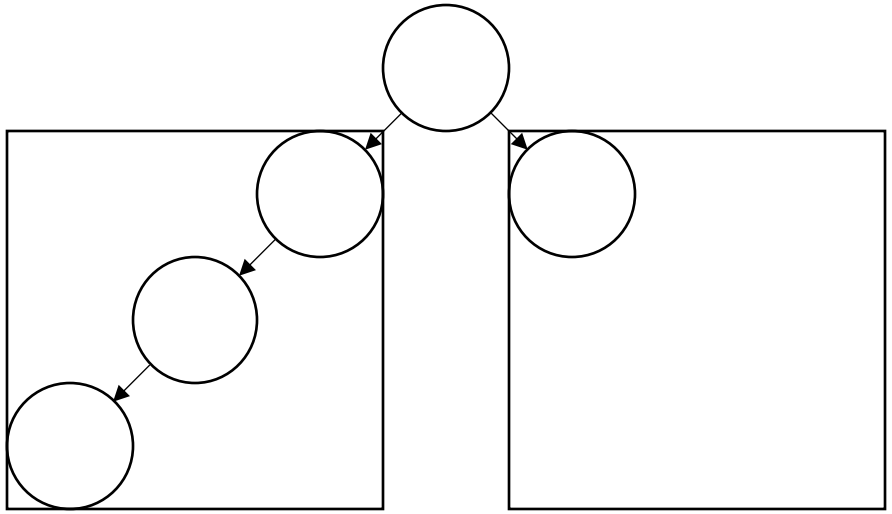
Insert

- … we already know how to insert into a BST

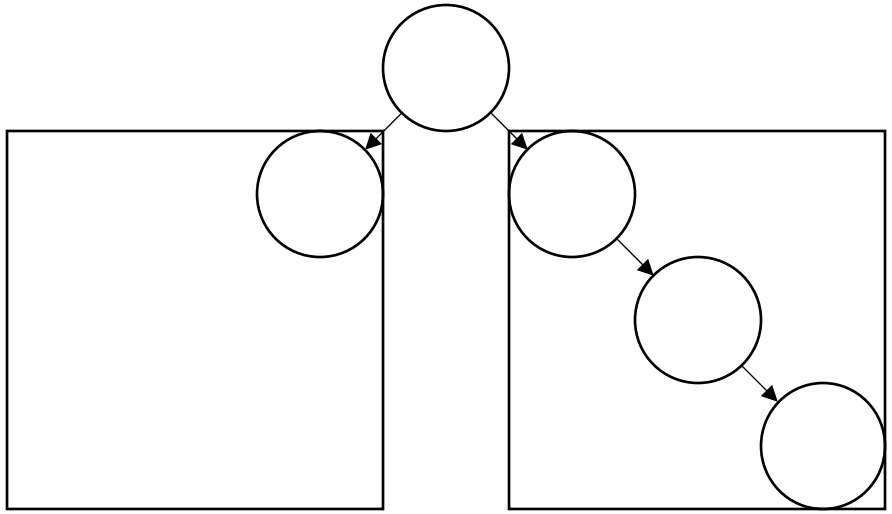
- For an AVL tree, we need to insert, check for imbalance, then balance tree
  - Insert node
  - Calculate balance factor
  - If balance factor is greater than 1 or less than -1,
  - Rotate to correct for imbalance

- BUT how do we know which way to rotate?
  - Left
  - Right

Insert

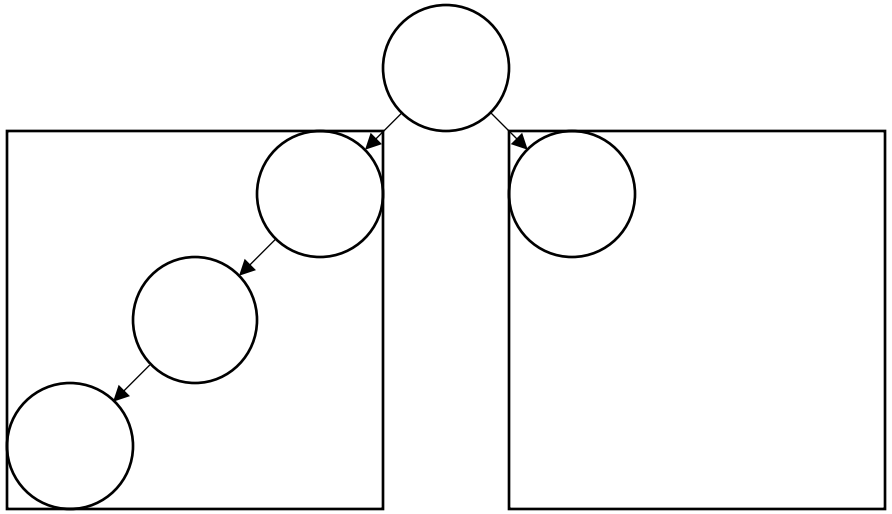


LST = 2, RST = 0
BF = 2
Imbalance = left

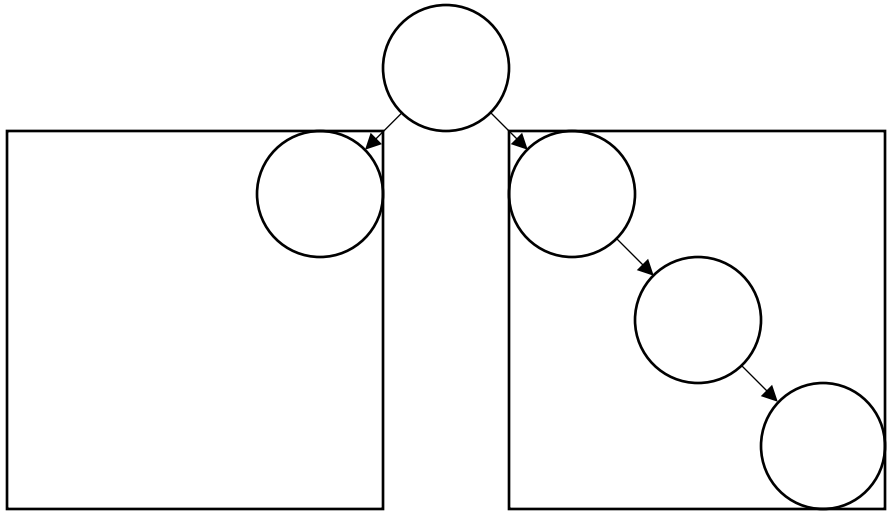


LST = 0, RST = 2
BF = -2
Imbalance = right

Insert



LST = 2, RST = 0
BF = 2
Imbalance = left



LST = 0, RST = 2
BF = -2
Imbalance = right

Positive BF = left imbalance

Negative BF = right imbalance

Insert

Positive BF = left imbalance

Negative BF = right imbalance

```
IF balance factor > 1
     THEN PRINT left imbalance
IF balance factor < -1
     THEN PRINT right imbalance
```

Insert

- Okay, lets try it …

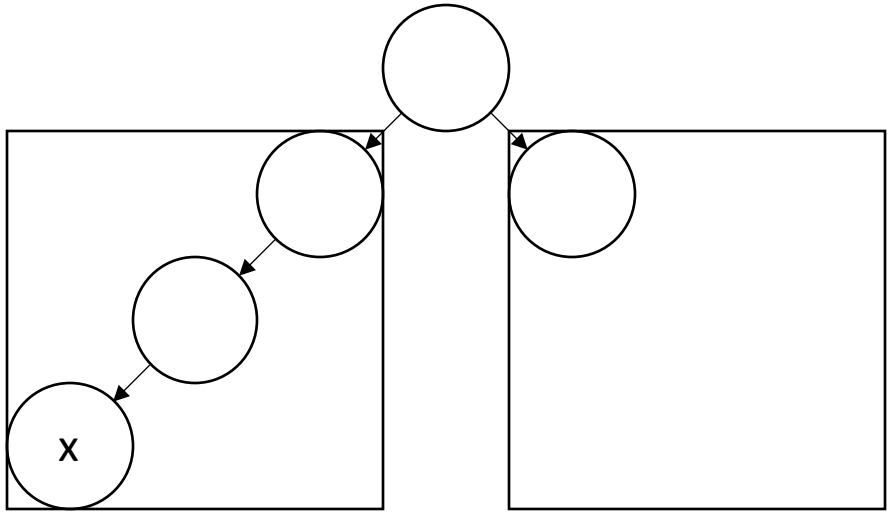
Insert

- … we already know how to insert into a BST

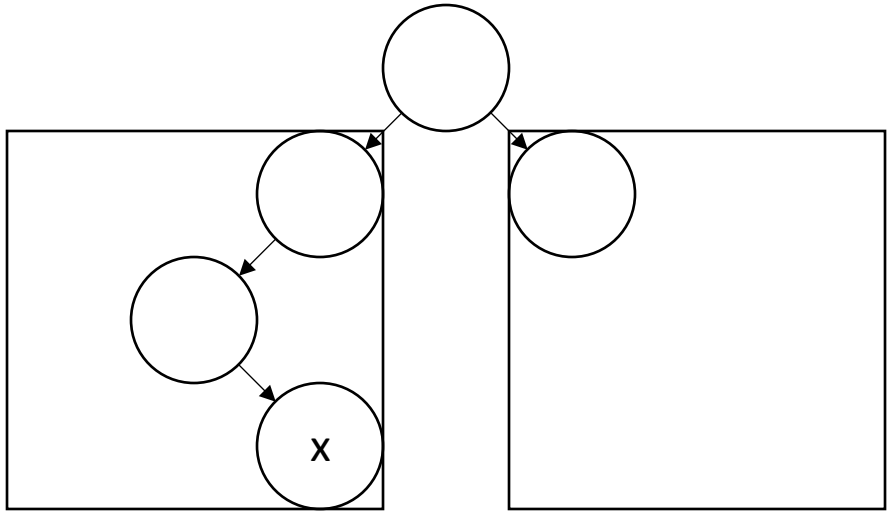
- For an AVL tree, we need to insert, check for imbalance, then balance tree
  - Insert node
  - Calculate balance factor
  - If balance factor is greater than 1 or less than -1,
  - Rotate to correct for imbalance

- BUT how do we know which way to rotate?
  - Left
  - Right
  - Left-right
  - Right-left

Insert

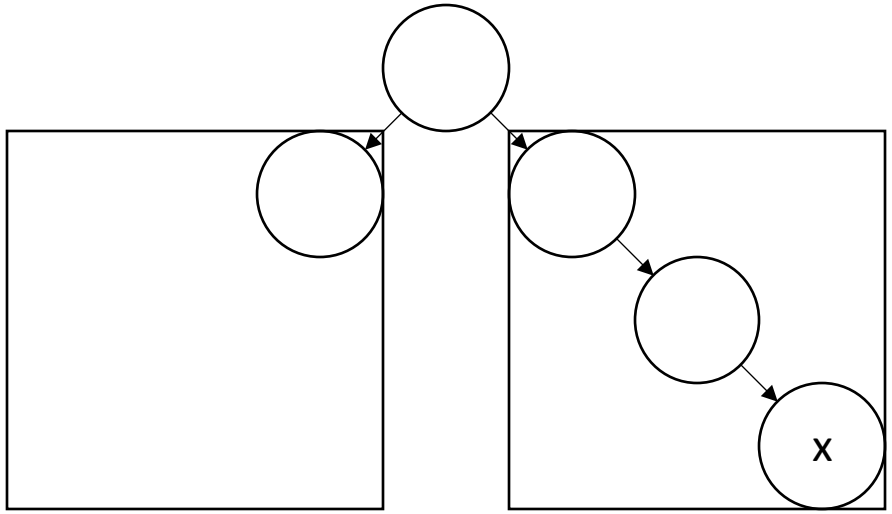


x < cRoot.left
Imbalance = left

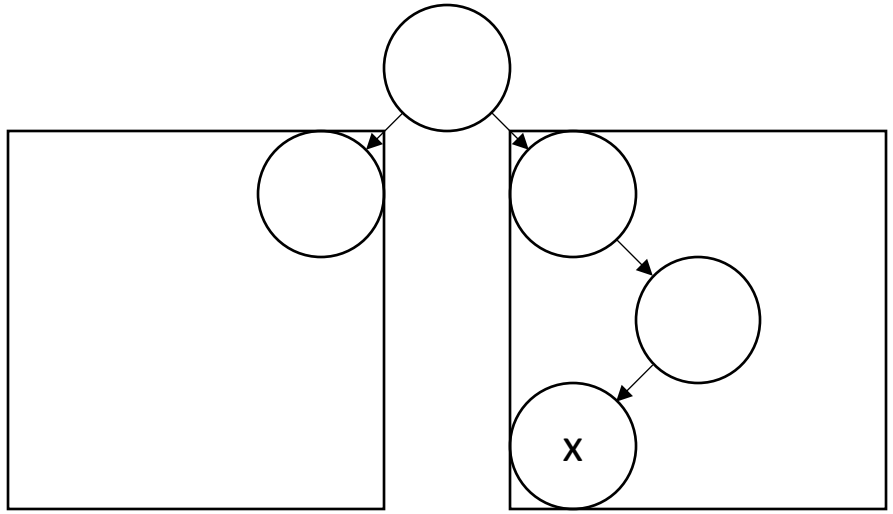


x > cRoot.left
Imbalance = left-right

Insert



x > cRoot.right
Imbalance = right



x < cRoot.right
Imbalance = right-left

Insert

Positive BF = left imbalance

Negative BF = right imbalance

```
IF balance factor > 1
     THEN PRINT left imbalance
IF balance factor < -1
     THEN PRINT right imbalance
```

Insert

```
IF balance factor > 1
      IF x < cRoot.left
            THEN PRINT left imbalance
      ELSE IF x > cRoot.left
            THEN PRINT left-right imbalance

IF balance factor < -1
      IF x > cRoot.right
            THEN PRINT right imbalance
      ELSE IF x < cRoot.right
            THEN PRINT right-left imbalance
```

Insert

- Okay, lets try it …

Insert

- … we already know how to insert into a BST

- For an AVL tree, we need to insert, check for imbalance, then balance tree
  - Insert node
  - Calculate balance factor
  - If balance factor is greater than 1 or less than -1,
  - Rotate to correct for imbalance

- BUT how do we know which way to rotate?
  - Left
  - Right
  - Left-right
  - Right-left

Insert

- Remember:
  - Left imbalance = right rotation
  - Right imbalance = left rotation
  - Left-right imbalance = left-right rotation
  - Right-left imbalance = right-left rotation

- … So …

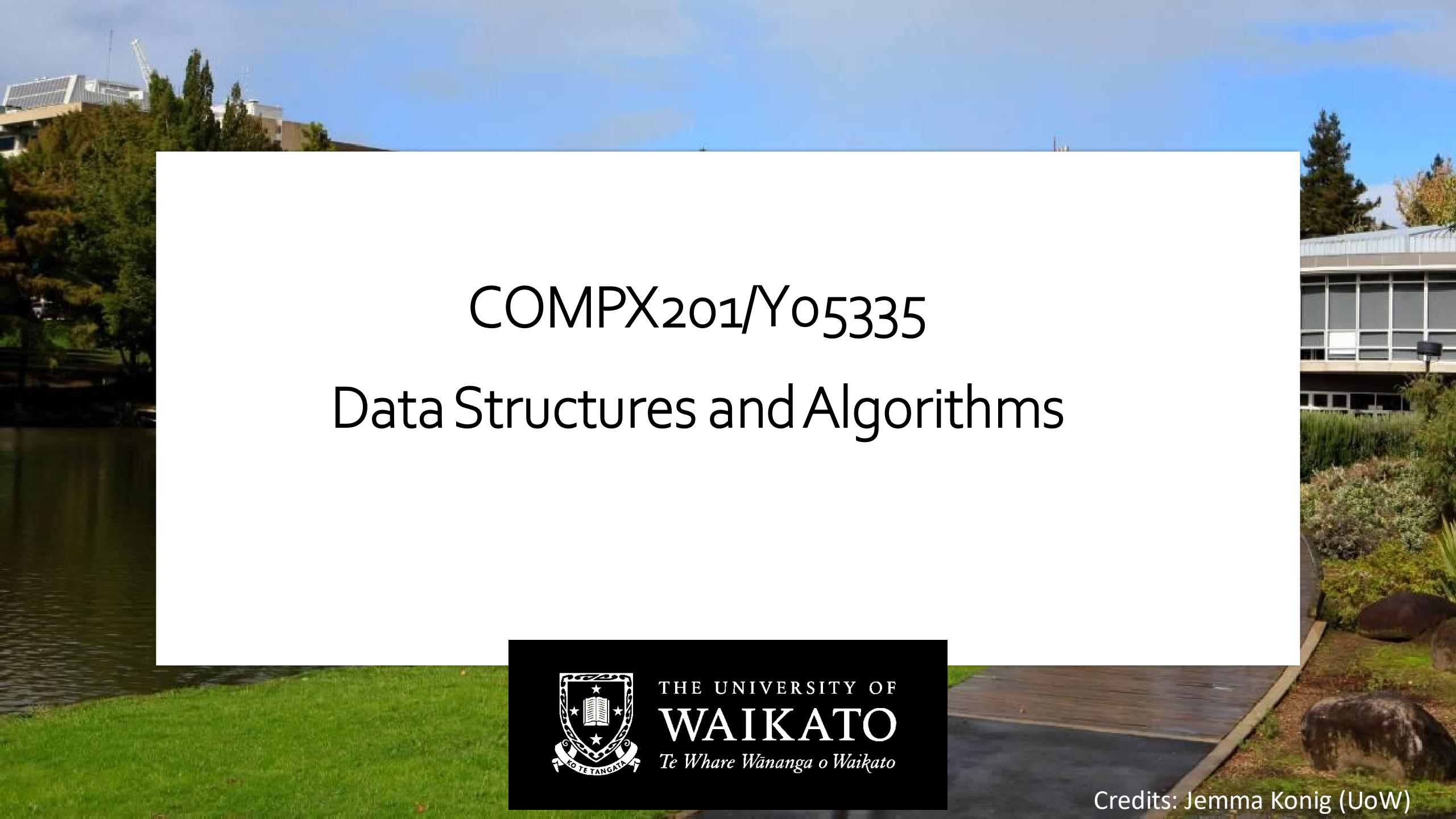
Insert

```
IF balance factor > 1
      IF x < cRoot.left
            RETURN right rotation
      ELSE IF x > cRoot.left
            SET cRoot.left = left rotation
            RETURN right rotation

IF balance factor < -1
      IF x > cRoot.right
            RETURN left rotation
      ELSE IF x < cRoot.right
            SET cRoot.right = right rotation
            RETURN left rotation
```

Insert

- Okay, let's try it ...



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)