# Data Structures

**Giri Narasimhan**
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

# Binary Tree vs BST

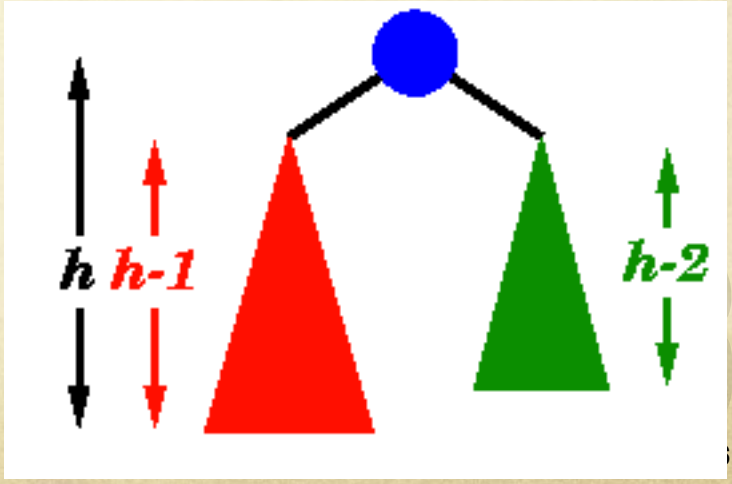
| | BinaryTree | BinarySearchTree |
|---|---|---|
| insert(x) | O(1) | O(h) |
| remove(x) | O(1) | O(h) |
| contains(x) | O(N) | O(h) |

- Main Problem:
  - Height of a BST, h = O(n)
  - A bad sequence of inserts can mess up the BST
  - OR, a bad sequence of deletes can mess up the BST
  - Making tree highly imbalanced and later searches very expensive

# A good BST

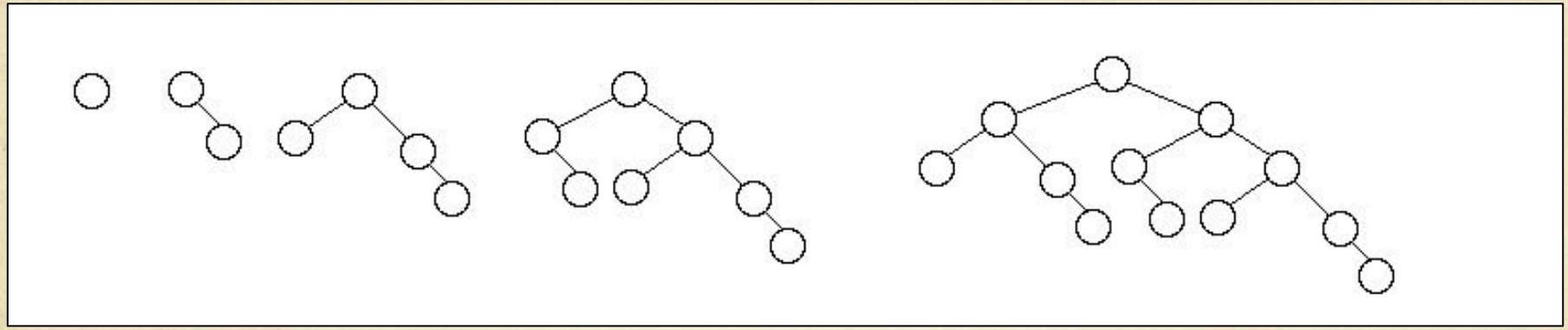
- A good BST has height h as small as possible.
- **Fact**: A binary tree with height h has $\leq 2^h - 1$ nodes
- **Fact**: A binary tree with n nodes has height $\geq \log_2(n+1)$
- A good BST should have
  - height = ceil($\log_2(n+1)$)
  - Height of subtrees are equal
- **Redefine**: A good BST has
  - Heights of subtrees off by $\leq 1$
  - at every node in tree
  - Problem: how to maintain it?

Does not work … Why?

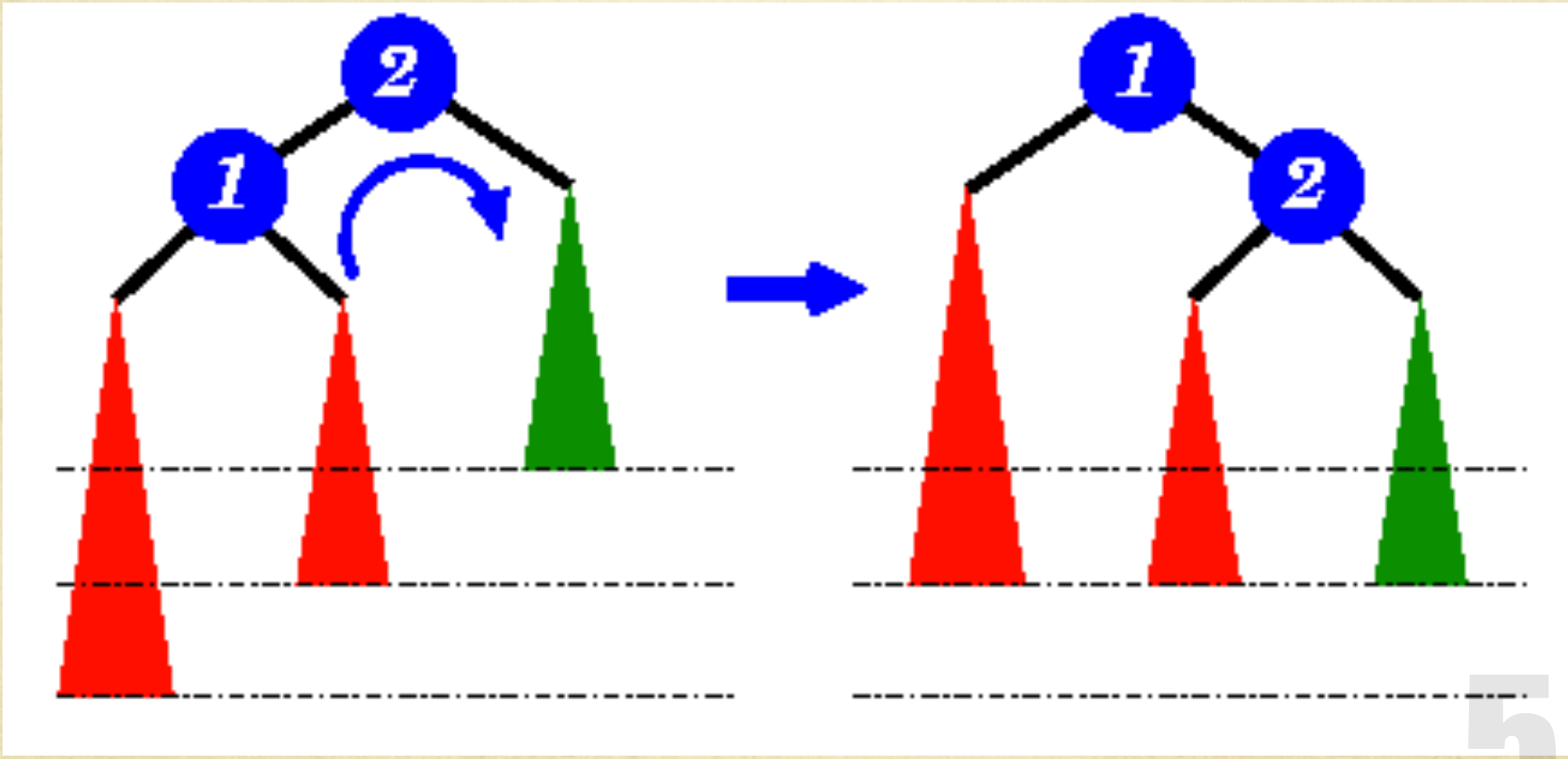



https://www.cs.auckland.ac.nz/software/AlgAnim/fig/AVL_bal.gif

# Worst-Case AVL Trees



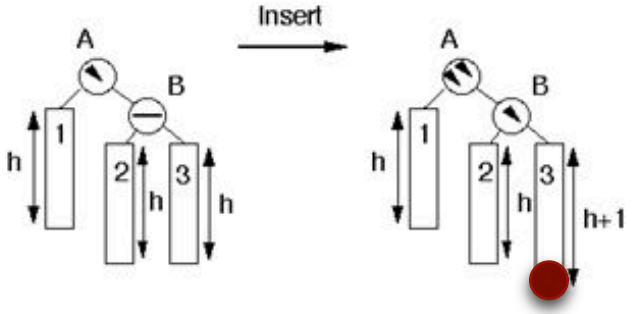
# Rebalance on Imbalance



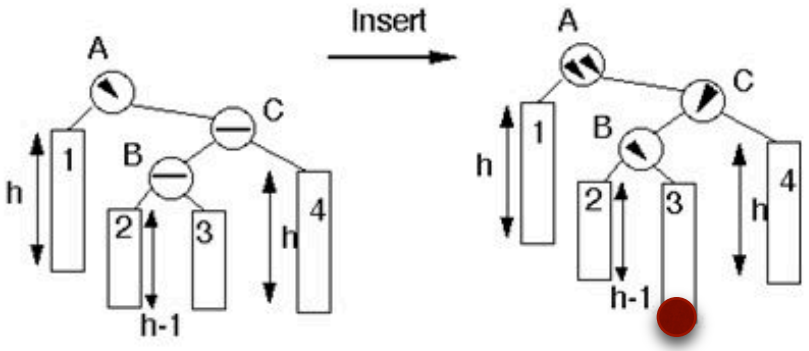

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/fig/AVL_case1.gif

# AVL Tree Insert

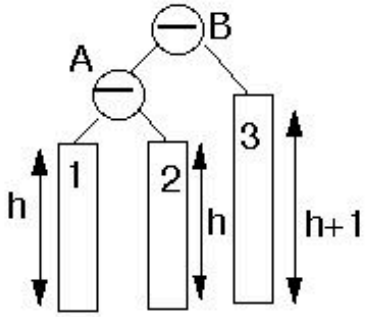
Insert into subtree 3



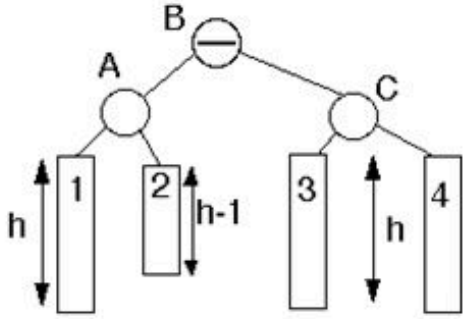

Insert into subtree 2 or 3




Perform single rotation




Perform double rotation

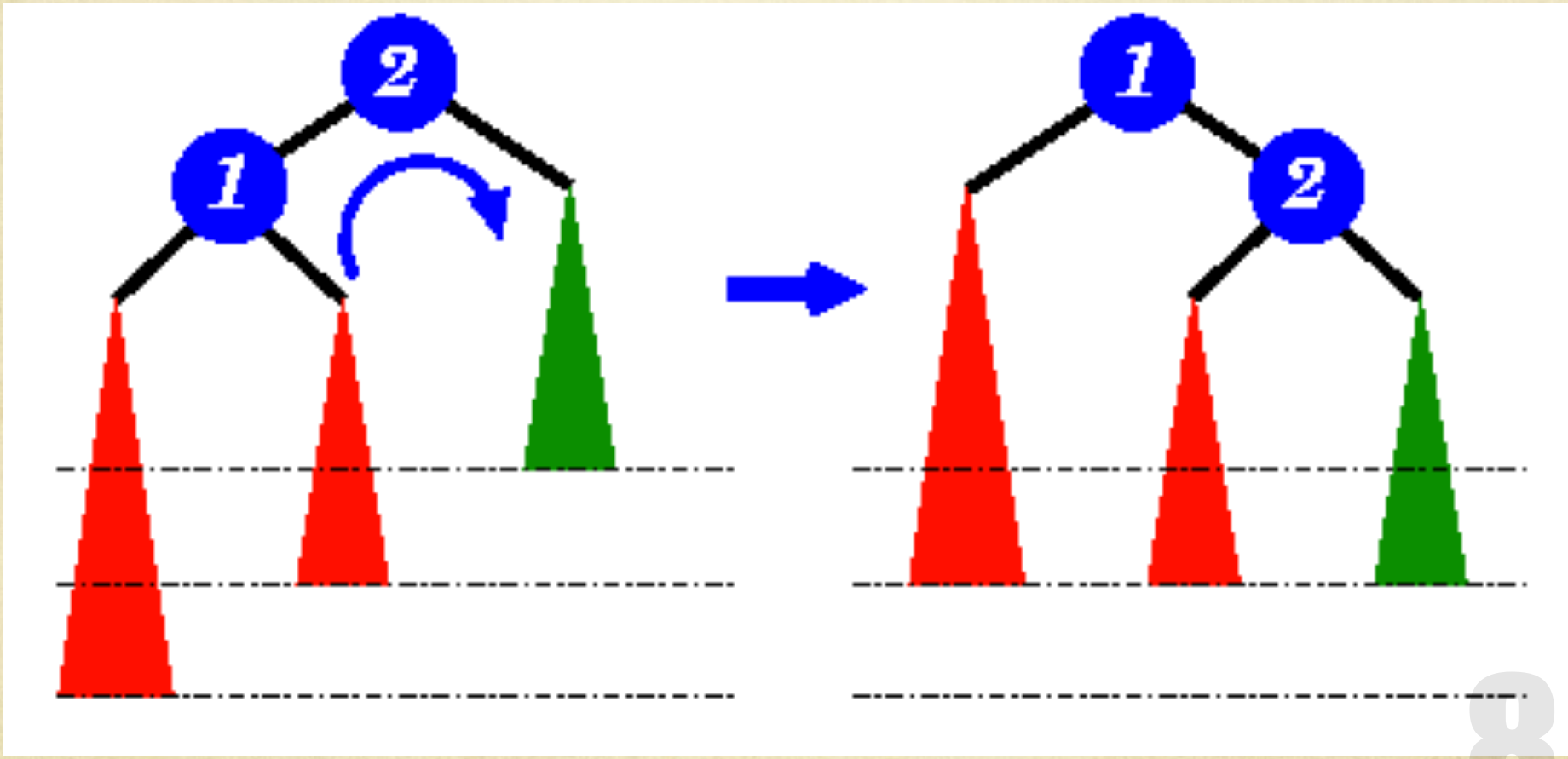



http://images.slideplayer.com/26/8338219/slides/slide_4.jpg

# AVL Tree Animation

- Maintain balance info at every node
- Insert:
  - Insert into BST and then rebalance all nodes on the way to the root of the tree
  - https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
- Delete:
  - Delete from BST and then rebalance all nodes on the way to the root of the tree
  - https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
- Also https://www.youtube.com/watch?v=FNeL18KsWPc

# Rebalance on Imbalance




https://www.cs.auckland.ac.nz/~jmor159/PLDS210/fig/AVL_case1.gif

# AvlNode

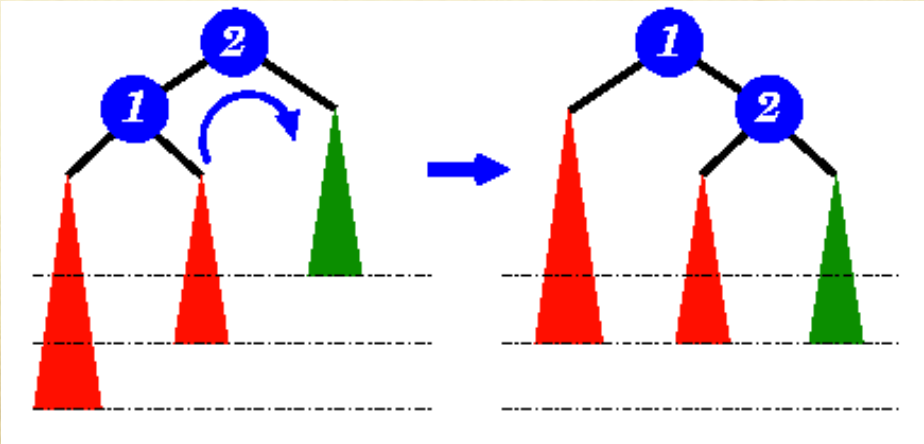
```
private static class AvlNode<AnyType>
  {
        // Constructors
    AvlNode( AnyType theElement )
    {
       this( theElement, null, null );
    }

    AvlNode( AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt )
    {
       element  = theElement;
       left     = lt;
       right    = rt;
       height   = 0;
    }

    AnyType          element;      // The data in the node
    AvlNode<AnyType>  left;         // Left child
    AvlNode<AnyType>  right;        // Right child
    int              height;       // Height
  }
```

# AVL Trees – same as BST

```
// ******************PUBLIC OPERATIONS**********
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// boolean contains( x )  --> Return true if x is present
// boolean remove( x )    --> Return true if x was present
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order
// ******************ERRORS*************************
// Throws UnderflowException as appropriate
```

# Single Rotation

```
private AvlNode<AnyType>
rotateWithLeftChild(AvlNode<AnyType> k2)
    {
        AvlNode<AnyType> k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        k2.height = max(height(k2.left),height(k2.right)) + 1;
        k1.height = max( height( k1.left ), k2.height ) + 1;
        return k1;
    }
```



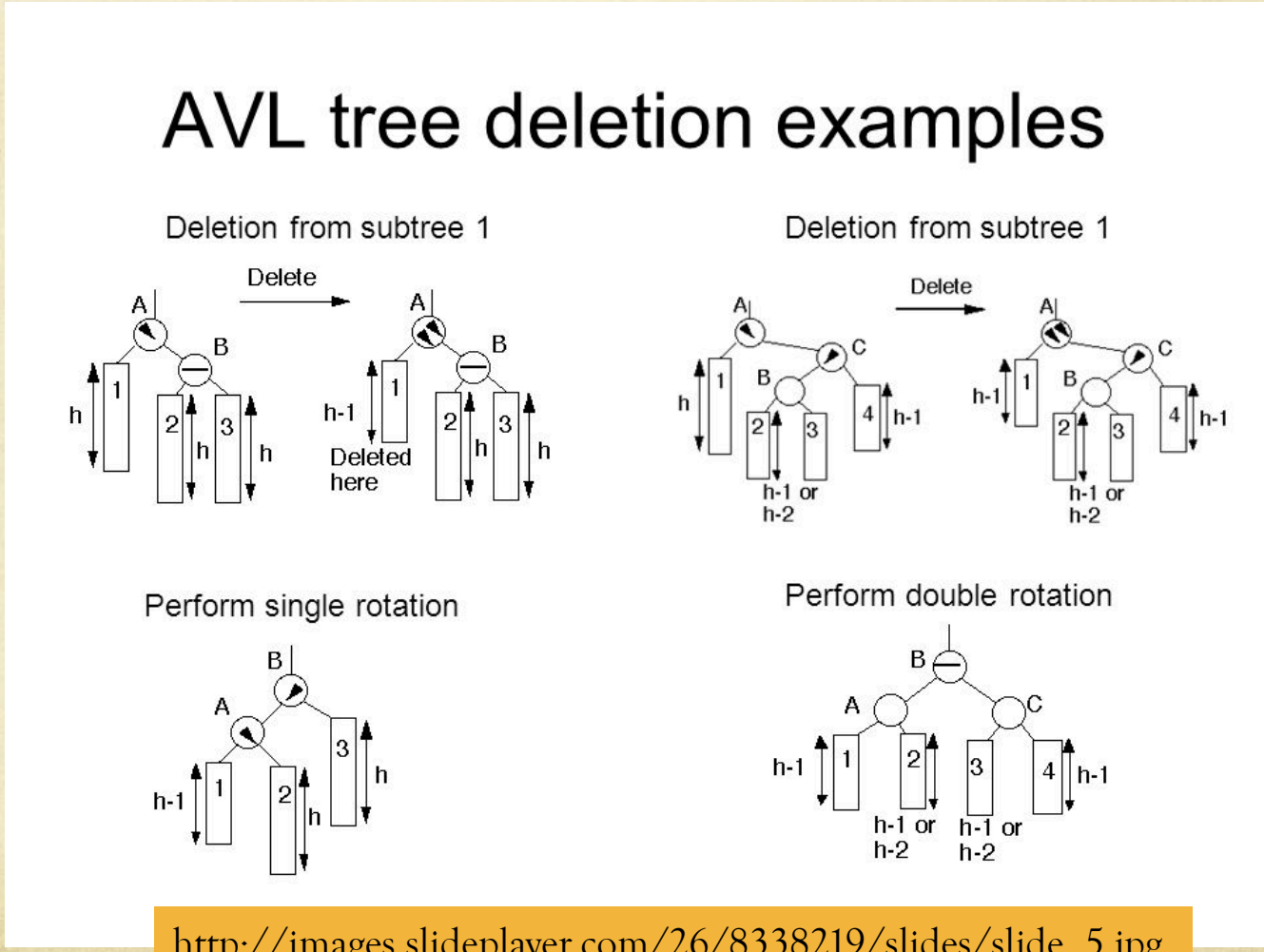
# Balance in AVL Trees

```
// Assume t is either balanced or within one of being balanced
private AvlNode<AnyType> balance( AvlNode<AnyType> t )
   {
      if( t == null )
         return t;

      if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )
         if( height( t.left.left ) >= height( t.left.right ) )
            t = rotateWithLeftChild( t );
         else
            t = doubleWithLeftChild( t );      // double rotation
      else
      if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )
         if( height( t.right.right ) >= height( t.right.left ) )
            t = rotateWithRightChild( t );
         else
            t = doubleWithRightChild( t ); // double rotation

      t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
      return t;
   }
```

# Insert in AVL Trees

```
public void insert(AnyType x) {root = insert( x, root ); } // non-recursive

private AvlNode<AnyType> insert( AnyType x, AvlNode<AnyType> t ) {
        // private recursive version of insert with extra parameter
        if( t == null )     // insert into empty tree; new node created
            return new AvlNode<>( x, null, null );

        int compareResult = x.compareTo( t.element );

        if( compareResult < 0 )          // insert into left subtree
            t.left = insert( x, t.left );  // add as left subtree after insertion
        else if( compareResult > 0 ) // insert into right subtree
            t.right = insert( x, t.right );
        else
            ;  // Duplicate; do nothing (to be modified as needed)
        return balance( t );             // rebalance if needed
    }
```

# AVL Tree Deletion




http://images.slideplayer.com/26/8338219/slides/slide_5.jpg

# Delete from AVL Tree

```
public void remove (AnyType x) {root = remove( x, root ); } // public non-recursive version

private AvlNode<AnyType> remove( AnyType x, AvlNode<AnyType> t ) {
      // private recursive version of insert with extra parameter
      if( t == null )
         return t;   // Item not found; do nothing

      int compareResult = x.compareTo( t.element );

      if( compareResult < 0 )                    // remove from left subtree
         t.left = remove( x, t.left );           // add as left subtree after removal
      else if( compareResult > 0 )               // remove from right subtree
         t.right = remove( x, t.right );
      else if( t.left != null && t.right != null ) // Two children
      {
         t.element = findMin( t.right ).element;
         t.right = remove( t.element, t.right );
      }
      else                                        // At most one child – easy case
         t = ( t.left != null ) ? t.left : t.right;
      return balance( t );                        // rebalance if needed
   }
```

# Major Difference: AVL - BST

- Height of BST Tree = h = O(n)
- Height of AVL Tree = h = O(log n)

| | BST | AVL Trees |
|---|---|---|
| insert(x) | O(h) | O(log N) |
| remove(x) | O(h) | O(log N) |
| contains(x) | O(h) | O(log N) |

# Static vs Dynamic Structures

# Time Complexity of List Operations

| | ArrayList | LinkedList |
|---|---|---|
| get(idx) | O(1) | O(N) |
| set(idx, x) | O(1) | O(N) |
| add(idx,x) | O(N) | O(N) |
| remove(idx) | O(N) | O(N) |
| addBefore(p,x) | | O(1) |
| Remove(p) | | O(1) |

# Linear Data Structures

- Lists
  - ArrayList
  - LinkedList
- Stacks & Queues
- Sorted Lists
  - List is sorted according to some key value in the data
  - Inserts and deletes must maintain sorted order
  - Search is an important operation
  - Implementations
    - ArrayList
    - LinkedList

# SortedList: Time Complexity

| | ArrayList | LinkedList |
|---|---|---|
| add(x) | O(N) | O(N) |
| remove(x) | O(N) | O(N) |
| find(x) | O(log N) | O(N) |

| | ArrayList | LinkedList |
|---|---|---|
| add(x) after find(x) | O(N) | O(1) |
| remove(x) after find(x) | O(N) | O(1) |
| find(x) | O(log N) | O(N) |

Tradeoffs!

| | BinaryTree | BinarySearchTree | AVL Trees |
|---|---|---|---|
| insert(x) | O(1) | | O(log N) |
| remove(x) | O(1) | | O(log N) |
| contains(x) | O(N) | | O(log N) |

# Static vs Dynamic Structures

- Static data structures
  - Build data structures once
  - Assume no (or few) updates and lots of queries
  - Use Static Structure such as SortedList or BST
- Dynamic data structures
  - Data structure with lots of updates and no/few queries
    - Stacks and queues
  - Data structure with lots of updates & lots of queries
  - Answers to queries depends on current state
  - E.g., Stacks and Queues