# CS 5010: Programming Design Paradigms

## Fall 2017

## Lecture 8: Data Structures and Algorithms II

Acknowledgement: lecture notes inspired by course material prepared by UW faculty members Z. Fung and E. McCarthy.

Tamara Bonaci
t.bonaci@northeastern.edu

# Administrivia

- Assignment 5 due on Monday, October 30 by 6pm
- Code walkthroughs on Tuesday, October 31 in your regular code walk sessions
- Next assignment – in pairs → watch our for more information on Piazza

[Picture credit: http://robertsrestaurantpaso.com/wp-content/uploads/2016/12/blog2.jpg]

# Agenda – Algorithms and Data Structures 2

- Some comments and hints about Assignments 4 and 5
- Trees
  - Tree Traversals
  - Binary trees
  - Search tree ADT
  - Balanced trees and AVL trees
- Maps and Sets in Java
- Hashing and Hash Functions in Java
  - HashTables
  - Collisions, probing and chaining

Algorithms and Data Structures 2

# ASSIGNMENTS 4 & 5 – COMMENTS AND HINTS

# Assignment 4 – A Parser

- Given:
  - Historical knowledge about the neighborhood
  - A list of desired candy for every child
- Find a way to traverse a neighborhood to get all the candy from the list

# Assignment 4 – A Parser

- Halloween Trick-or-treating:
  - Different houses give out different kinds of treats (candy) – a language/grammar
  - Kids have very particular preferences what kinds of candy would they like to get – a token stream
  - Neighborhood traversal – a parse tree

# Assignment 5 – Background

- Online transactions – important to verify:
  - An identity of a client
  - A content of the request

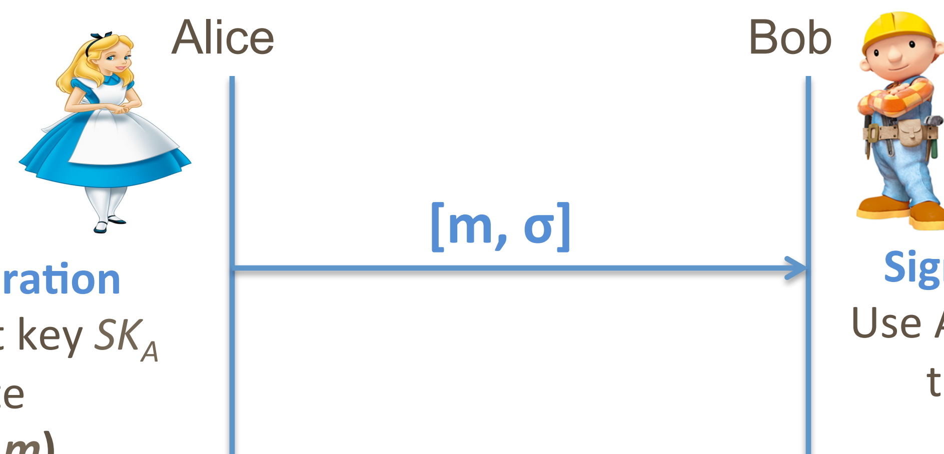
- Both verifications can be done using digital signatures

# Assignment 5 – Your Task

- Simulate digital signature verification process for some fictional bank and its unique clients
- Simulator takes several input arguments:
  - Number of unique bank clients
  - Number of distinct transactions
  - Fraction of invalid messages
  - Output files

# Quick Introduction to Digital Signatures

Digital signature – a *public-key* mechanism to provide data integrity and authentication




**Signature generation**

Use Alice's secret key $SK_A$ to compute

$\boldsymbol{\sigma = sig(SK_A, m)}$

**Signature verification**

Use Alice's public key $PK_A$ to verify message

$\boldsymbol{ver(PK_A, m, \sigma)}$

If $ver(PK_A, m, \sigma)$ = true accept message

[Picture credit: pintrest.com, hitentertainment.com]

# The RSA Digital Signature Scheme

RSA digital signature – relies on the difficulty of factoring the product of large prime numbers

Every digital signature scheme consists of three main steps:

1. Key generation
2. Signature generation
3. Signature verification

# RSA Digital Signature – Key Generation

*The keys for the RSA digital signature are generated in the same way as the keys for RSA encryption:*

1. Generate two distinct large primes $p$ and $q$
2. Compute the product $n = pq$
3. Compute the Euler totient function $\phi(n) = (p - 1)(q - 1)$
4. Randomly generate an integer e that satisfies:

$$\gcd(e, n) = 1$$
$$\gcd(e, \phi(n)) = 1$$

5. Compute $d$ such that $de \equiv 1 \bmod \phi(n)$

Public key PKA = (e,n)
Private key SKA = (d,n)

# RSA Digital Signature – Signature Generation

Alice generates a signature on a message $m$ using her private key $SK_A$:

$$\sigma = sig(SK_A, m) = m^d \bmod n$$

# RSA Digital Signature – Signature Verification

1. Bob uses Alice's public key $PK_A$, to compute:
   $\sigma^e \bmod n = (m^d)^e \bmod n = m^{ed} \bmod n = m'$

2. He compares the received message $m$ and $\sigma^e \bmod n$
   - If $m = \sigma^e \bmod n \rightarrow$ message accepted

# RSA Digital Signature – Summary

Alice

Bob

[m, σ = $m^d$ mod n]

**Signature generation**

Use Alice's secret key $SK_A$ to compute

$\sigma = m^d \bmod n$

**Signature verification**

Use Alice's public key $PK_A$ to verify message

$ver(PK_A, m, \sigma) = \sigma^e \bmod n$

If $\sigma^e \bmod n = m$ accept message

[Picture credit: pintrest.com, hitentertainment.com]

Algorithms and Data Structures 2

# DATA COLLECTIONS

# Data Collections?

Collection of chewed gums

Collection of pens

Collection of cassette tapes

Collection of old radios

## What is a data collection?

Shoes collection

Star wars collection

Cars collection

[Pictures credit: http://www.smosh.com/smosh-pit/articles/19-epic-collections-strange-things ]

# Data Collections?

- Data collection - an object used to store data (think *data structures*)
  - Stored objects called elements
  - Some typically operations:
    - `add()`
    - `remove()`
    - `clear()`
    - `size()`
    - `contains()`
- Some examples: ArrayList, LinkedList, Stack, Queue, Maps, Sets, Trees

Algorithms and Data Structures 2

# TREES

# Trees

[Pictures credit: https://static1.squarespace.com/]

# Trees

- **Tree** - a directed, acyclic structure of linked nodes
  - **Directed** - one-way links between nodes
  - **Acyclic** - no path wraps back around to the same node twice

- Can be defined recursively:
  - A tree is either:
    - Empty(null), or
    - A **root** node that contains:
      - Data
      - A **left** subtree
      - A **right** subtree

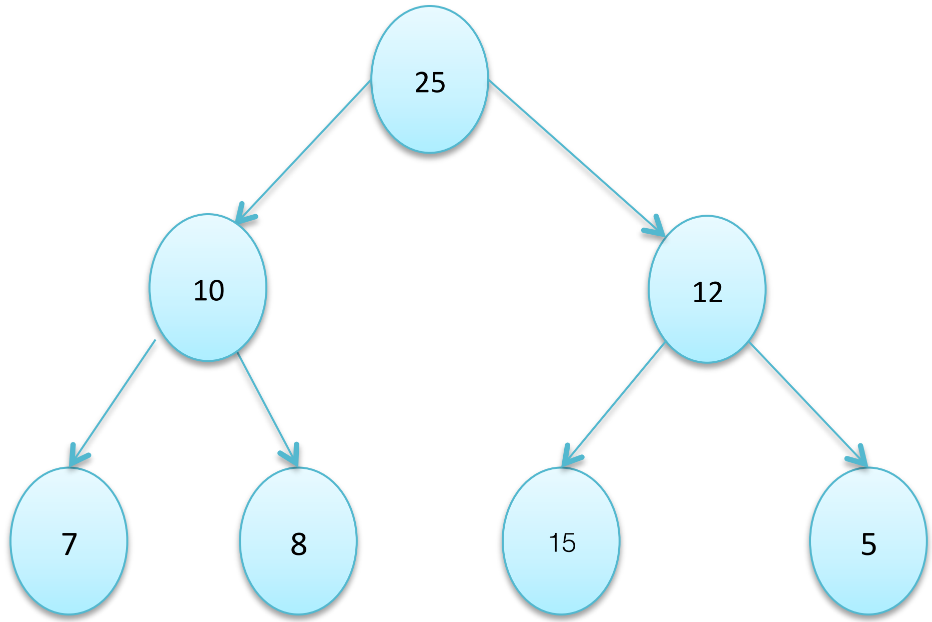
      (The left and/or right subtree could be empty)

# Trees Terminology

- Node - an object containing a data value and left/right children
- Root - topmost node of a tree
- Subtree – a smaller tree of nodes on the left or right of the current node
- Parent - a node above the left and right subtrees, that both subtrees are connected to
- Child - a root of each subtree
- Sibling - a node with a common parent
- Leaf - a node that has no children
- Branch - any internal node; neither the root nor a leaf
- Level or depth - length of the path from a root to a given node
- Height - length of the longest path from the root to any node

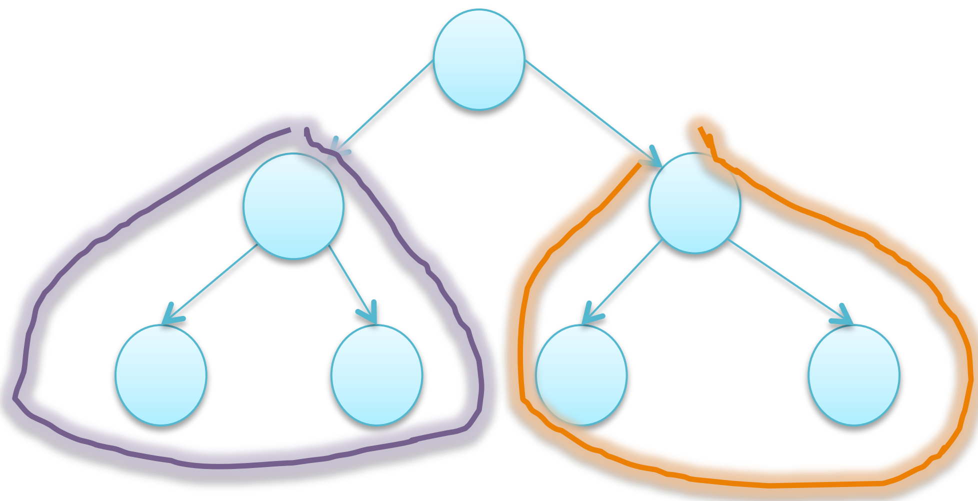
# Trees Terminology Example

- **Nodes**: {25, 10 ,12, 7, 8, 15, 5}
- **Root**: 25
- **Subtrees:** {10, 7, 8} and {12, 15, 5}
- **Parents**: 10 → {7, 8}, 12 → {15, 5}, 25→ {10, 12}
- **Children**: {10, 12, 7, 8, 15, 5}
- **Siblings**: {7, 8}, {15, 5} and {10, 12}
- **Leaves:** {7, 8, 15, 5}
- **Height:** 3



# Binary Trees

- Binary tree – a tree in which no node can have more than two children



# Binary Tree Implementation

- A basic `BinaryNode` object stores:
  - Data,
  - Link to the left child
  - Link to the right child
- Multiple nodes can be linked together into a larger tree

```
class BinaryNode{
//Friendly data; accessible by other package routines
   Object element;
   BinaryNode left;
   BinaryNode right;
}
```

# Example: Class `StrigTreeNode`

StringTreeNode class

```
// A StringTreeNode object is one node in a binary tree of String
    public class StringTreeNode{
    public String data; // data stored at this node
    Public StringTreeNode left; // reference to left subtree
    Public StringTreeNode right; // reference to right subtree

    // Constructs a leaf node with the given data
    Public StringTreeNode(String data){
     this(data, null, null);
    }
    // Constructs a branch node with the given data and links
    Public StringTreeNode(String data, StringTreeNode left, StringTreeNode right){
     this.data = data;
     this.left = left;
     this.right = right;
    }
}
```

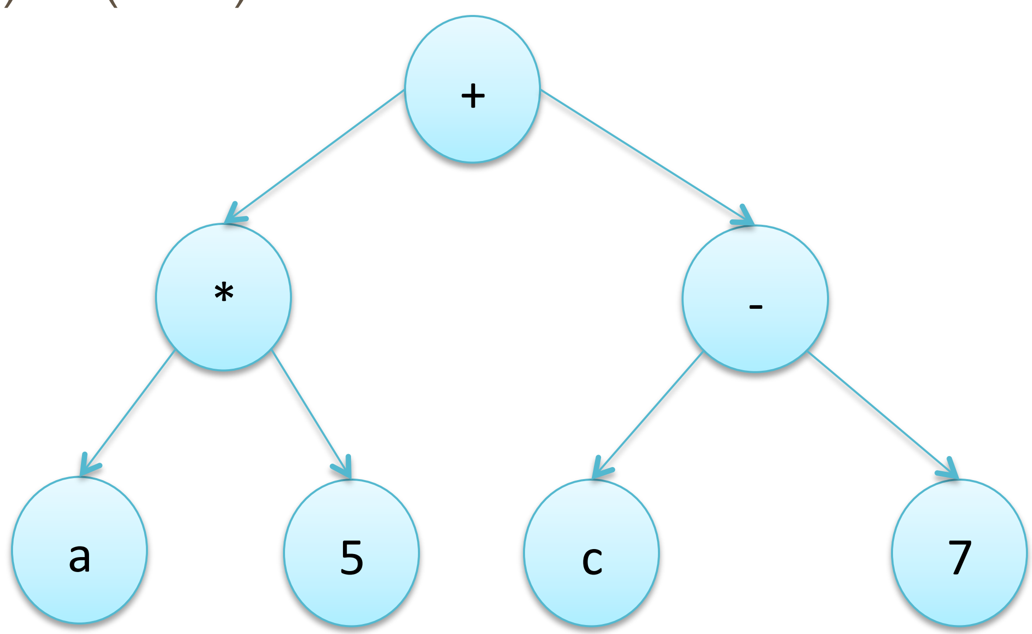
# Example: Class `StringTree`

```
// An StringTree object represents an entire binary tree of
String.
public class StringTree{
    private StringTreeNode root;
    //some methods
}
```

- Observations:
  - We can only talk to the `StringTree`, not to the node objects inside the tree
  - Methods of the `StringTree` create and manipulate the nodes, their data and links between them

# Example: Expression Trees

- In an expression tree:
  - Leaves are operands (constants or variable names)
  - All other leaves are operators (unary or binary)
  - Example: (a * 5) + (c -7)

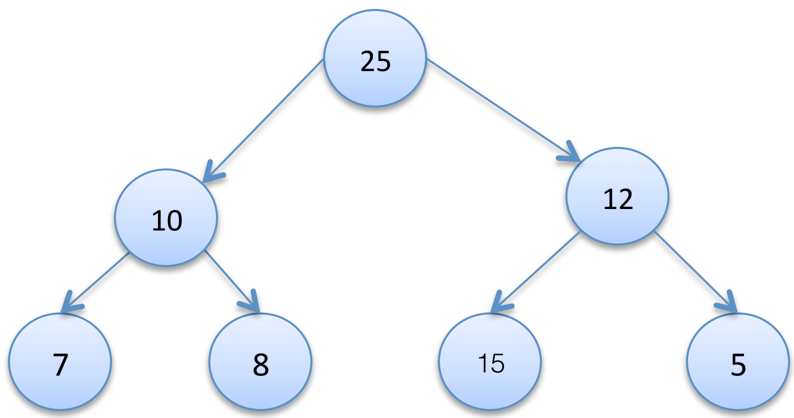


Algorithms and Data Structures 2

# TREE TRAVERSALS

# Searching an Element in a Tree

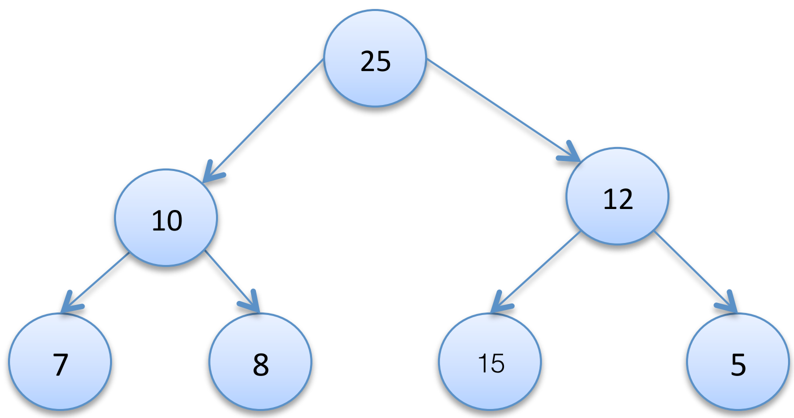
- Example: find element 15 in the given tree




- Possible approaches:
  - Depth-first search (DFS)
  - Breath-first search (BFS)

# Breath-First Search

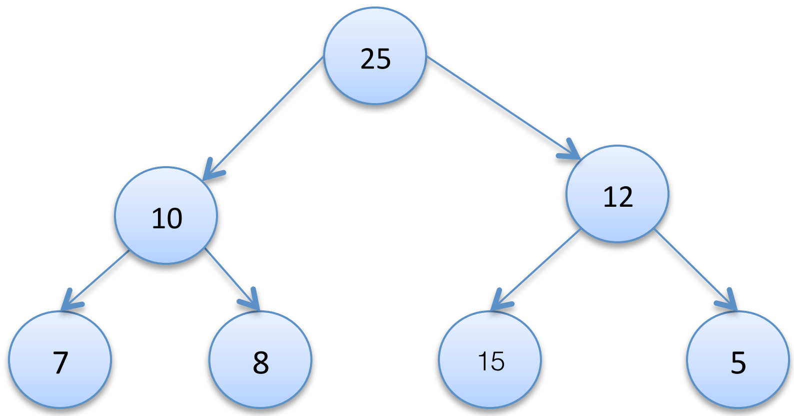
- Example: find element 15 in the given tree




- Traverse all of the nodes on the same level first, and then move on to the next (lower) level

# Depth-First Search

- Example: find element 15 in the given tree



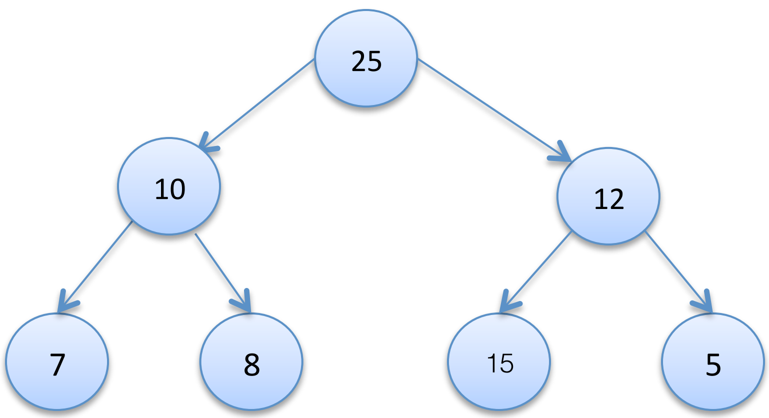

- Traverse one side of the tree all the way to the leaves, followed by the other side

# Tree Traversals

- Tree traversal - an examination of the elements of a tree
  - Used in many tree algorithms and methods
- Common orderings for traversals:
  - Pre-order – process root node, then its left/right subtrees
  - In-order – process left subtree, then root node, then right subtree
  - Post-order – process left/right subtrees, then root node

# Tree Traversals Example

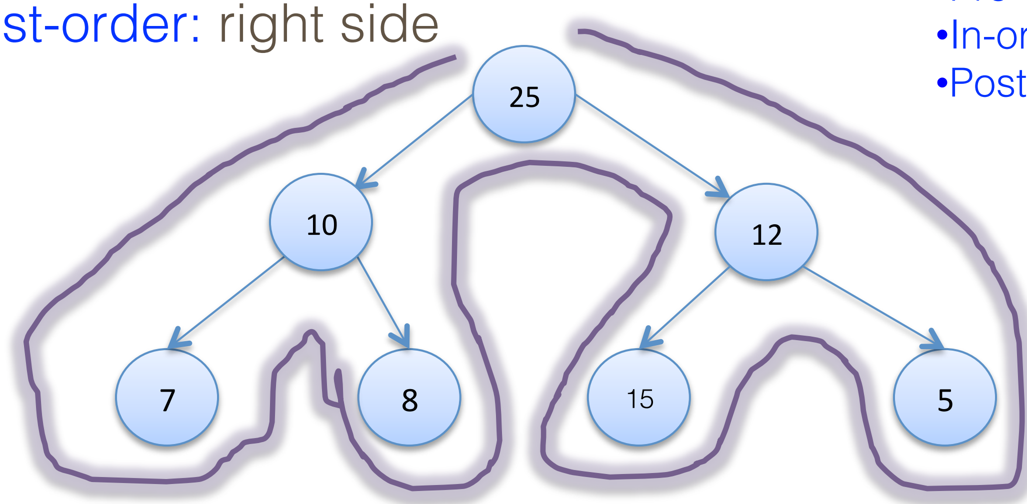
- Common orderings for traversals:
  - Pre-order – process root node, then its left/right subtrees
  - In-order – process left subtree, then root node, then right subtree
  - Post-order – process left/right subtrees, then root node




- Pre-order: 25 10 7 8 12 15 5
- In-order: 7 10 8 25 15 12 5
- Post-order: 7 8 10 15 5 12 25

# Tree Traversals Trick

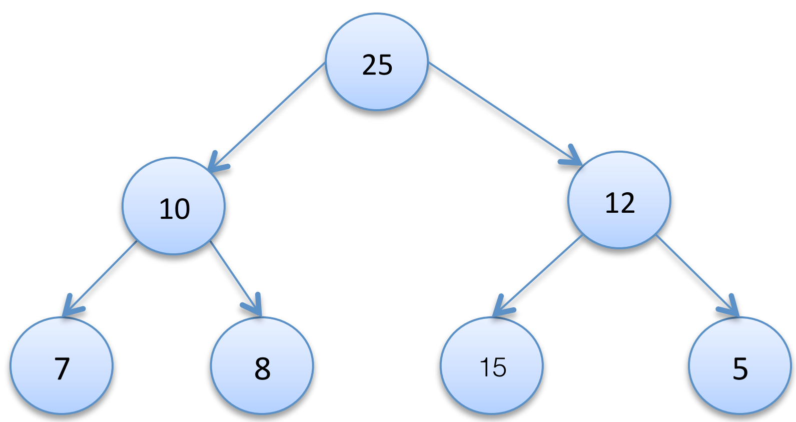
- To quickly generate a traversal, trace a path around the tree
- As you pass a node on the proper side, process it:
  - Pre-order: left side
  - In-order: bottom
  - Post-order: right side




- Pre-order: `25 10 7 8 12 15 5`
- In-order: `7 10 8 25 15 12 5`
- Post-order: `7 8 10 15 5 12 25`

# Example: Printing a Tree

- Assume we have some class `IntTree`
- Add a method `print` to the `IntTree` class that prints the elements of the tree, such that
  - Elements of a tree are separated by spaces
  - A node's left and right subtree should be printed before it
- Example: `tree.print(); //7 8 10 15 5 12 25`



# Example: Printing a Tree

```
// An IntTree object represents an entire binary tree of ints
public class IntTree{
    private IntTreeNode overallRoot; // null for an empty tree ...
     public void print(){
     print(overallRoot);
     System.out.println(); // end the line of output
     }
     private void print(IntTreeNode root){
      // (base case is implicitly to do nothing on null)
         if (root != null){
         // recursive case: print left, right, center
              print(overallRoot.left);
              print(overallRoot.right);
              System.out.print(overallRoot.data + " ");
         }
     }
 }
```

# Template for Tree Methods

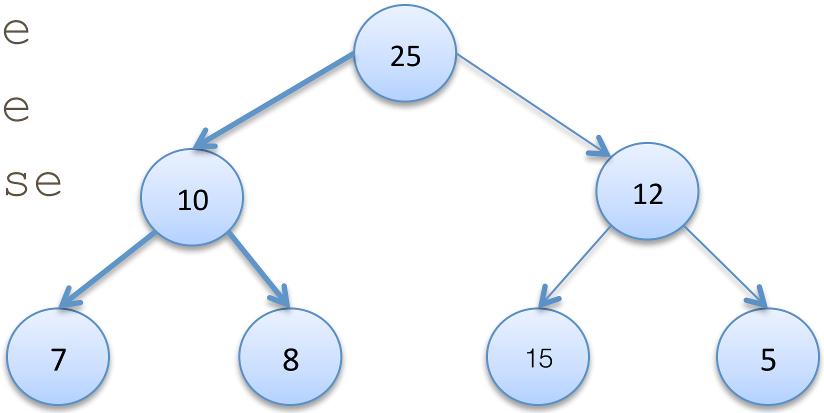
- Tree methods are often implemented recursively with a public/private pair
  - The private version accepts the root node to process

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...
    public type name(parameters) {
        name(overallRoot, parameters);
    }
    private type name(IntTreeNode root, parameters) {
        ...
    }
}
```

# Example: `contains()`

- Add a method `contains` to the `IntTree` class that searches the tree for a given integer, returning `true` if it is found.

- Example: If an `IntTree` variable `tree` referred to the tree below, the following calls would have these results:
  - `tree.contains(25)` → `true`
  - `tree.contains(12)` → `true`
  - `tree.contains(4)` → `false`
  - `tree.contains(77)` → `false`



# Example: contains()

```
// Returns whether this tree contains the given integer
public boolean contains(int value){
    return contains(overallRoot, value);
}
private boolean contains(IntTreeNode node, int value){
    if (node == null){
        return false; // base case: not found here
    }else if (node.data == value){
        return true; // base case: found here
    }else{
        // recursive case: search left/right subtrees
        return contains(node.left, value) || contains(node.right,
              value);
     }
}
```

Algorithms and Data Structures 2

# SEARCH TREE ADT

# Refresher: Binary Search

- Binary search – a search that finds a target value in a *sorted* data collection by successively eliminating half of the collection from consideration
- In the worst case how many elements will need to examined
- Example: Find value 25 in the array below:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value | -5 | 0 | 6 | 7 | 13 | 20 | 25 | 56 | 78 | 124 | 203 | 255 |

Min Mid Max

# Arrays.binarySearch()

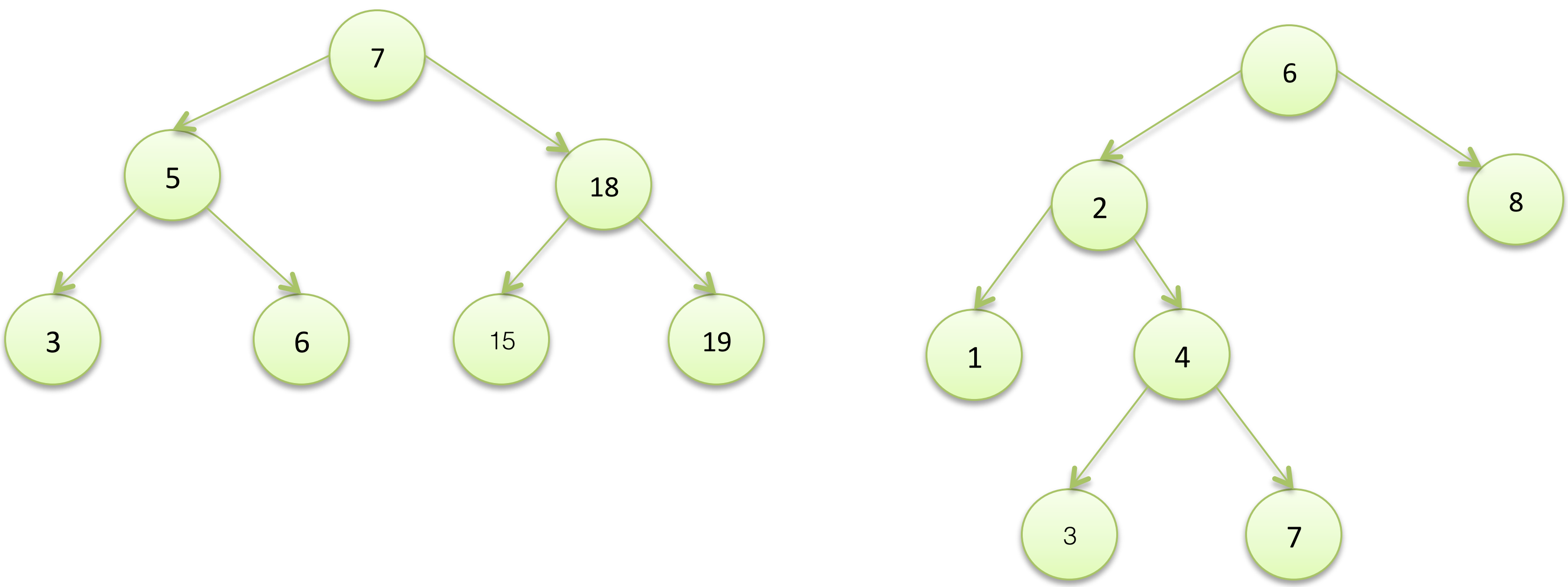
```
// searches an entire sorted array for a given value
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, value)

// searches given portion of a sorted array for a given value
// examines minIndex (inclusive) through maxIndex (exclusive)
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- If the value is found, `binarySearch()` returns the index
- If the value is *not* found, `binarySearch()` returns `-(insertionPoint + 1)`, where `insertionPoint` is the index where the element *would* have been, if it had been in the array in sorted order

# Binary Search Trees (BST)

- **Binary search tree** – a binary three that stores element in a sorted order
- Every non-empty node *X* of some BST has the property that:
  - Elements of *X*'s left subtree contain data smaller than *X*'s data
  - Elements of *X*'s right subtree contain data greater than *X*'s
  - *X*'s left and right subtrees are also binary search trees

# Example: Binary Search Trees?



# Class `BinarySearchTree`

```
public class BinarySearchTree <T extends Comparable <? super T>> {
    private static class BinaryNode<T> {
    //build binary node}

    public BinaryNode<T> root;

    public BinarySearchTree() {
        root = null;
    }

    public void makeEmpty() {
        root = null;
    }

    public boolean isEmpty()  {
        return root == null;
    }
```

# Class BinarySearchTree

```
public class BinarySearchTree <T extends Comparable <? super T>> {
    private static class BinaryNode<T> {
    //build binary node}

    public boolean contains(T x) {
        return contains(x, root);
    }

    public T findMin() {
        if(isEmpty()) throw new NullPointerException();
        return findMin(root).element;
    }

    public T findMax() {
        if(isEmpty()) throw new NullPointerException();
        return findMax(root).element;
    }
```

# Class BinarySearchTree

```
public class BinarySearchTree <T extends Comparable <? super T>> {
    private static class BinaryNode<T> {
    //build binary node}

    public void insert(T x) {
        root = insert(x, root);
    }

    public void remove(T x) {
        root = remove(x, root);
    }
```

# Method `private contains(T x, BinaryNode<T> node)`

```
/**
* Internal method to find an item in a subtree.
* @param x is item to search for.
* @param node the node that roots the subtree.
* @return true if the item is found; false otherwise.
*/
private boolean contains(T x, BinaryNode<T> node) {
    if(node == null)
        return false;

    int compareResult = x.compareTo(node.element );
    if(compareResult < 0 )
        return contains(x, node.left );
    else if(compareResult > 0)
        return contains(x, node.right);
    else return true; // Match
}
```

# Methods `private findMin and findMax`

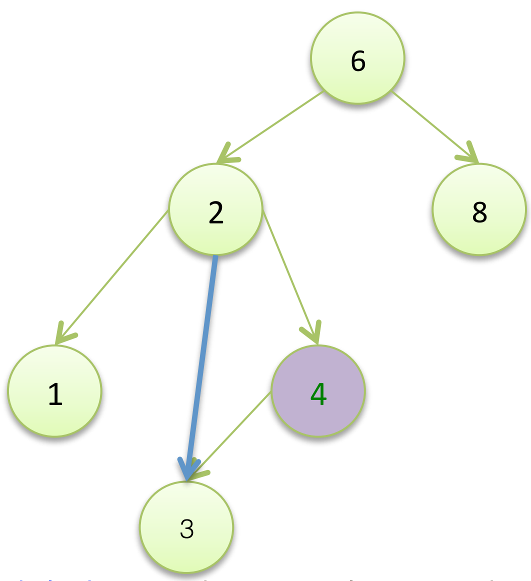
```
/** * Internal method to find the smallest item in a subtree.
* @param t the node that roots the subtree.
* @return node containing the smallest item.
*/
 private BinaryNode<T> findMin(BinaryNode<T> t) {
     if(t == null)
          return null;
     else if(t.left == null)
          return t;
     return findMin(t.left);
 }

/** * Internal method to find the largest item in a subtree.
* @param t the node that roots the subtree.
* @return node containing the largest item.
*/
private BinaryNode<T> findMax(BinaryNode<T> t) {
     if(t == null)
          return null;
     while(t.right != null)
          t = t.rigth;
     return t;
 }
```

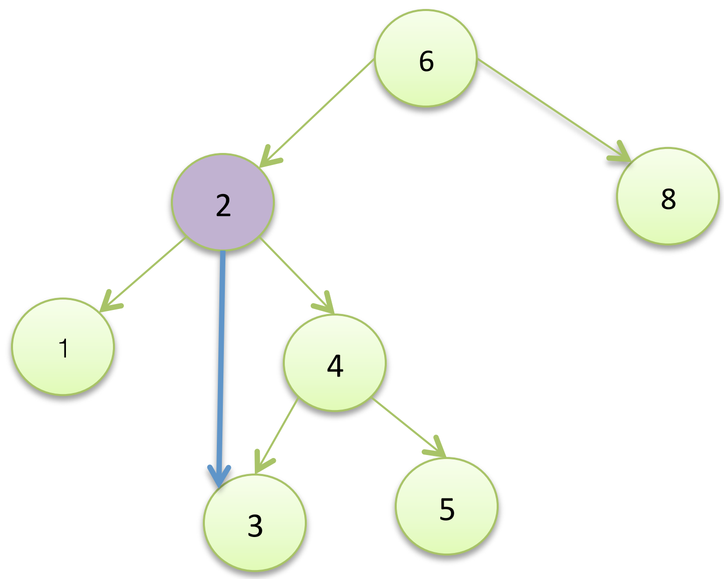
# Methods `private insert(T x, BinaryNode<T> t)`

```
/**
 * Internal method to insert into a subtree.
  * @param x the item to insert.
  * @param t the node that roots the subtree.
  * @return the new root of the subtree.
 */
 private BinaryNode<T> insert(T x, BinaryNode<T> node) {
     if(t == null) {
          return new BinaryNode<>(x, null, null);
     }

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

     if(compareResult < 0)
          node.left = insert(x, node.left);
     else if(compareResult > 0)
          node.right = insert(x, node.right);
     else
          return node; //duplicate, do nothing
}
```

# Methods `private remove(T x, BinaryNode<T> t)`

- Deletion – the hardest operation
- Once the node to delete has been found, we need to consider several possibilities:
  - Node is a leaf – can be deleted immediately
  - Node has one child – can be deleted after its parent adjusts a link to bypass it
  - Node has two children – replace data of that node with the smallest data of the right subtree



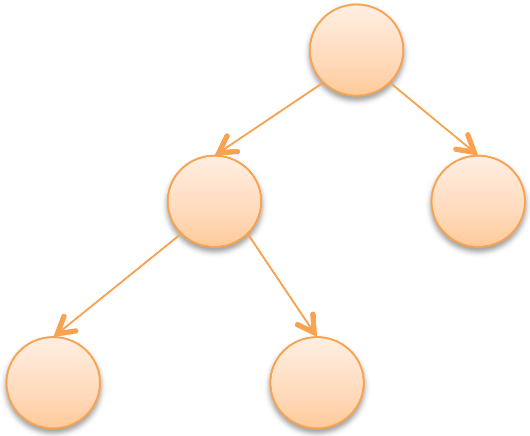

- Lazy deletion – when an element should be deleted, it is left in the tree, and merely marked as being deleted

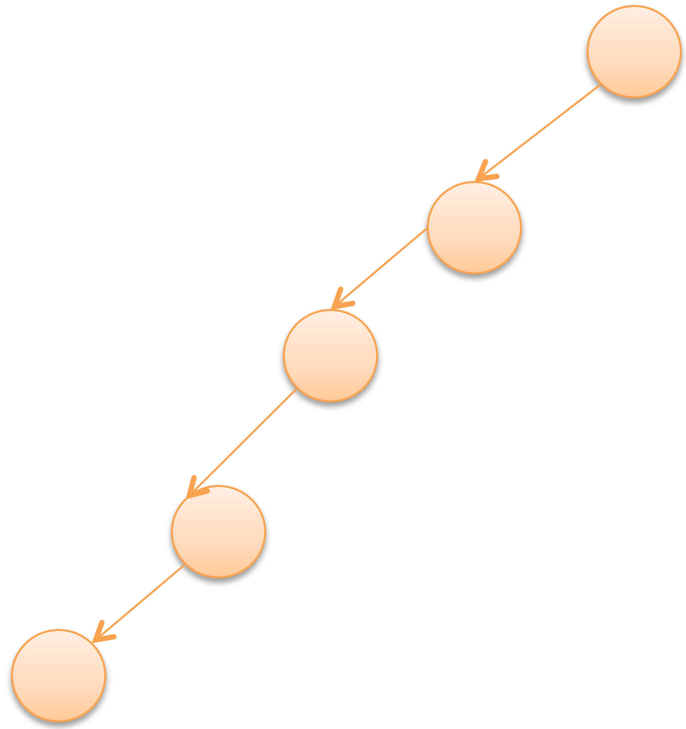
Algorithms and Data Structures 2

# BALANCED AND AVL TREES

# Tree Balance and Height

- If the same data can be represented multiple ways, what is best?





# Tree Balance and Height

- If the same data can be represented multiple ways, what is best?

- Height is key for how fast functions on our tree are!
- If we can structure the same data two different ways, we may want to choose a balanced structure (better for BSTs)

- Can we enforce balance?

# Tree Balance and Height

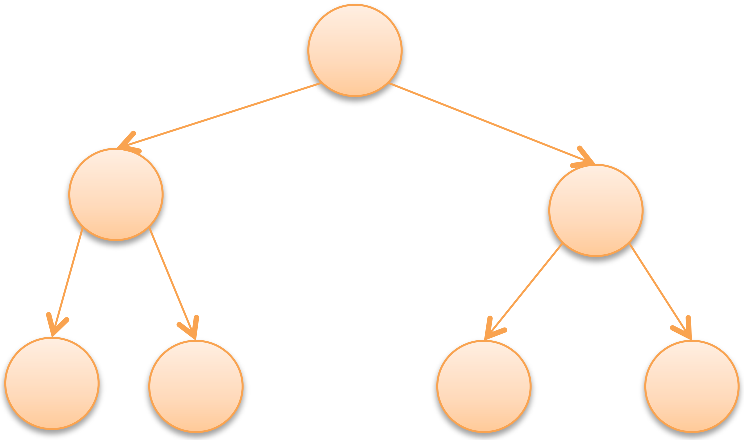
- How might we define balance?

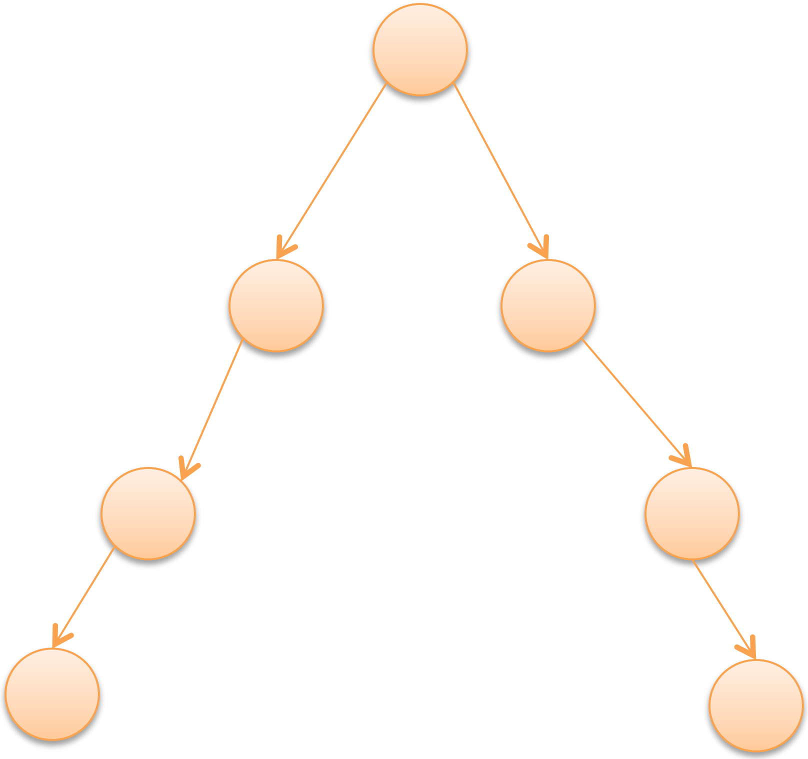
- If the heights of the left and right trees are balanced, the tree is balanced, so:

  Abs(height(left) – height(right))

- Anything wrong with this?

# Tree Balance and Height





# Tree Balance and Height

- Is is not enough for the root to be balanced
- All nodes must be balanced
- Ideally, our "balance" property should say:
  - For all nodes in the tree, level(left)= level(right)

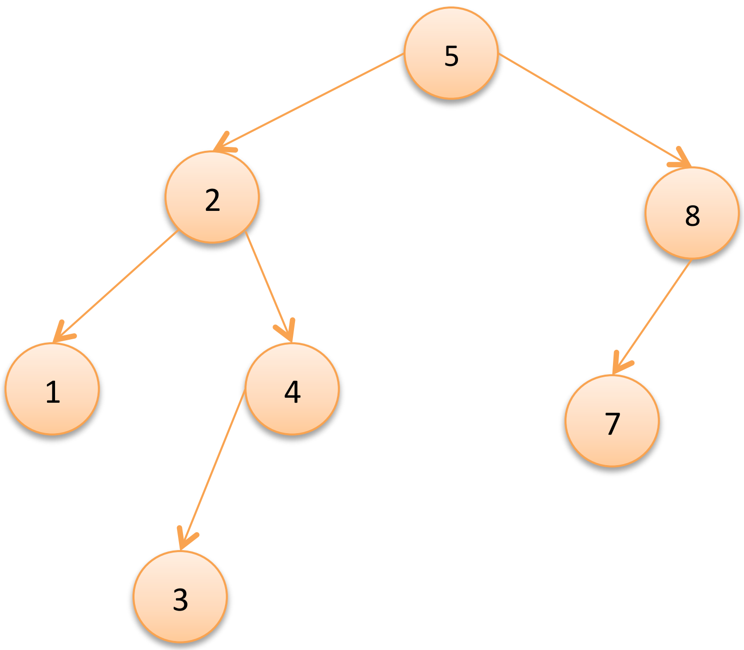
- What is the problem with this? Not always enforceable!

# AVL (Adelson-Velskii-Landis) Tree

- AVL tree – binary search tree with a balance condition (AVL condition):
  - The height of the left and right subtrees differ by at most 1
- The height of an empty tree defined to be -1
- All tree operations (exception insertion) can be performed in O(logN)

# Example: AVL Trees??

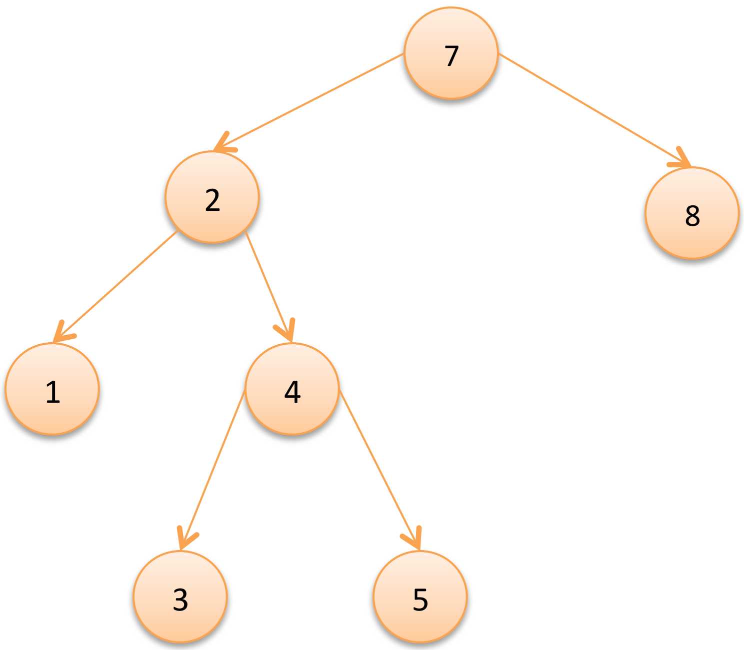
- Is this an AVL tree?




Yes!

# Example: AVL Trees??

- Is this an AVL tree?




No!

# AVL Operations

- Since AVL trees are also BST trees, they should support the same functionality:
  - `insert(T x, BinaryNode<T> node)`
  - `find(T x) – same as BST`
  - `delete(T x)`

- Problem: inserting the node could violate the AVL property
- Solution: the AVL property maintained as we add the node → simple tree modification, rotation

# AVL insert() Operation

- Possible insertions:
  1. Insertion into the left subtree of the left child of X
  2. Insertion into the right subtree of the left child X
  3. Insertion into the left subtree of the right child of X
  4. Insertion into the right subtree of the right child of X

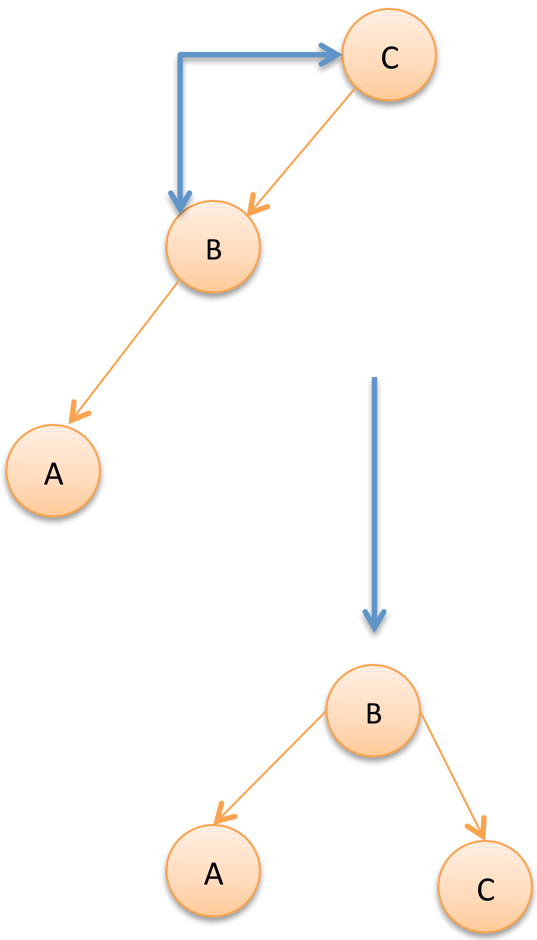
# Single and Double AVL Rotations

- In cases 1 and 4, the insertions occurs on the “outside”:
  - Left-left insertion (case 1)
  - Right-right insertion (case 2)
- Outside insertions can be “fixed” using single rotations

- Similarly, in cases 2 and 3, the insertions occur on the “inside”:
  - Left-to-right (case 2)
  - Right-to-left (case 3)
- Inside rotations require double rotations to fix imbalance

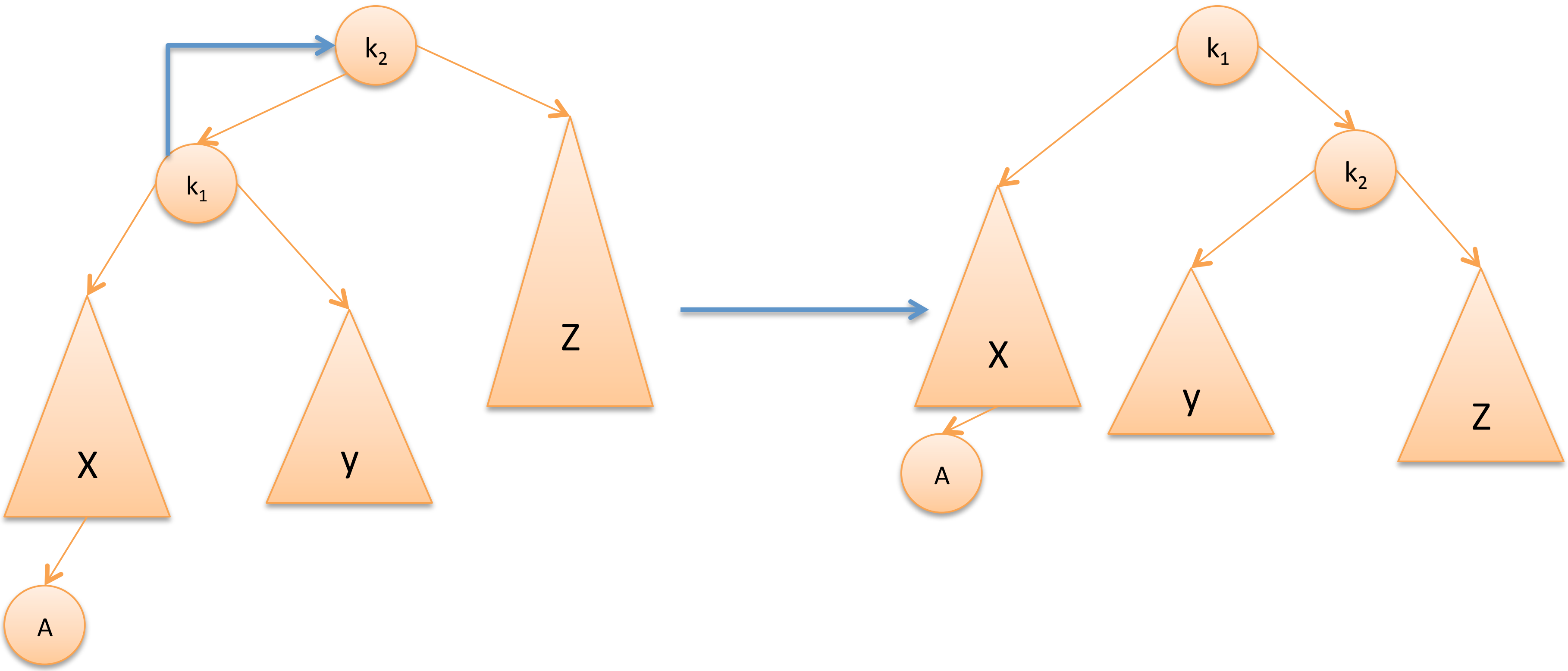
- Caution: with an insertion, a balance might not be off on the parent → it could be somewhere up the tree
-

# Left-to-left Insertion and AVL Rotation

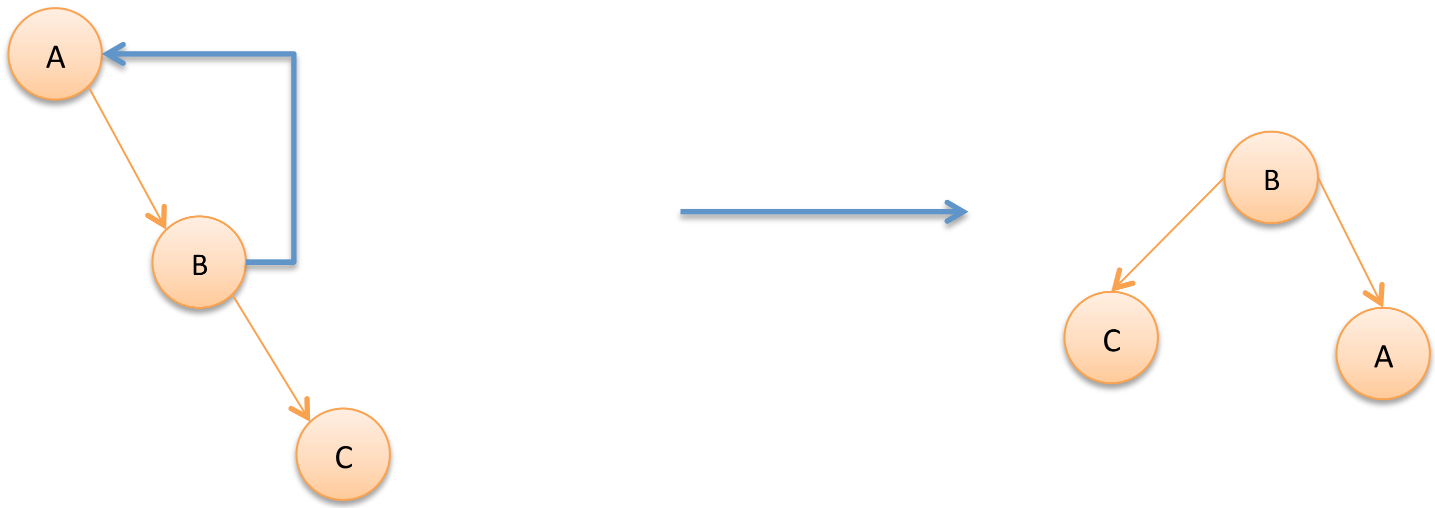



- Problem: not balanced anymore
- To correct this insertion – left rotation:
- B must become the root
- A must become the left child of B

# General Rotation for Left-to-left Insertion

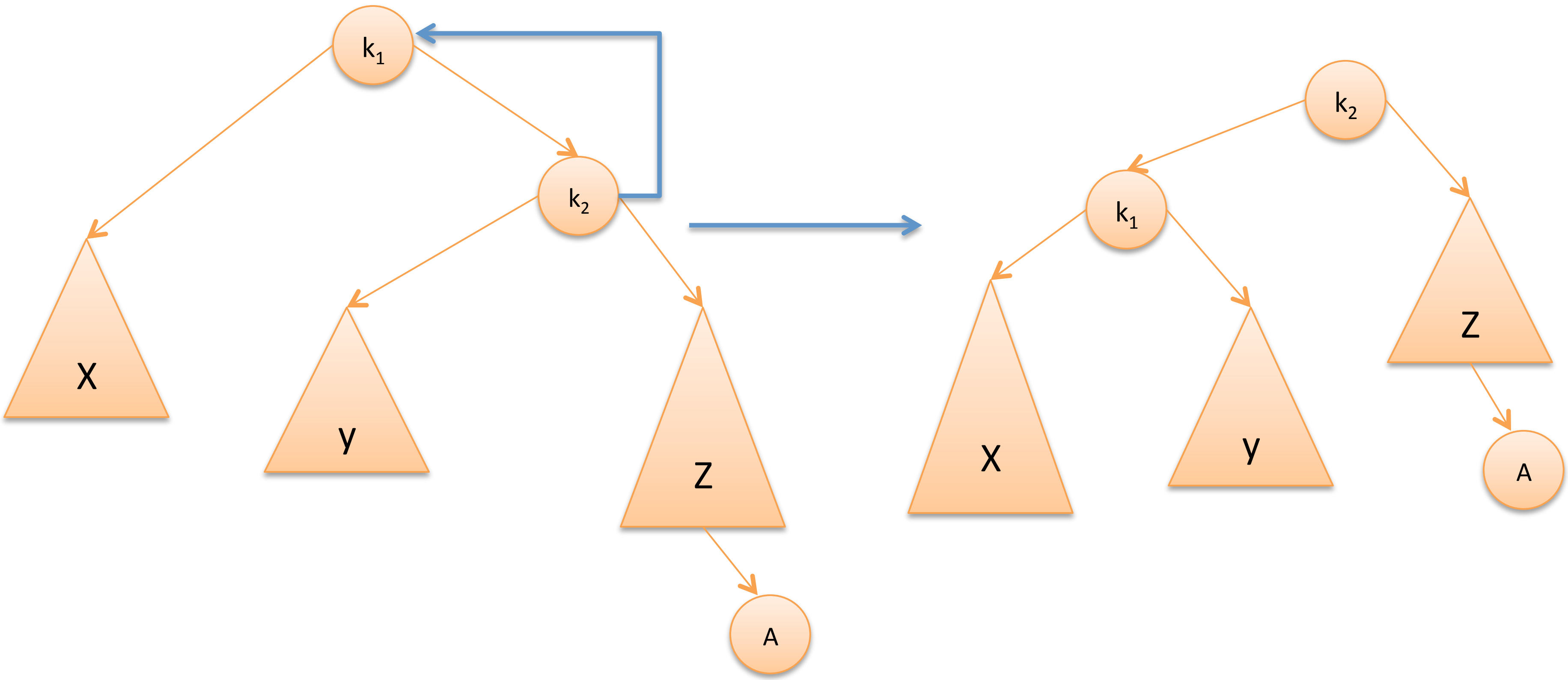


# Right-to-Right Insertion and AVL Rotation

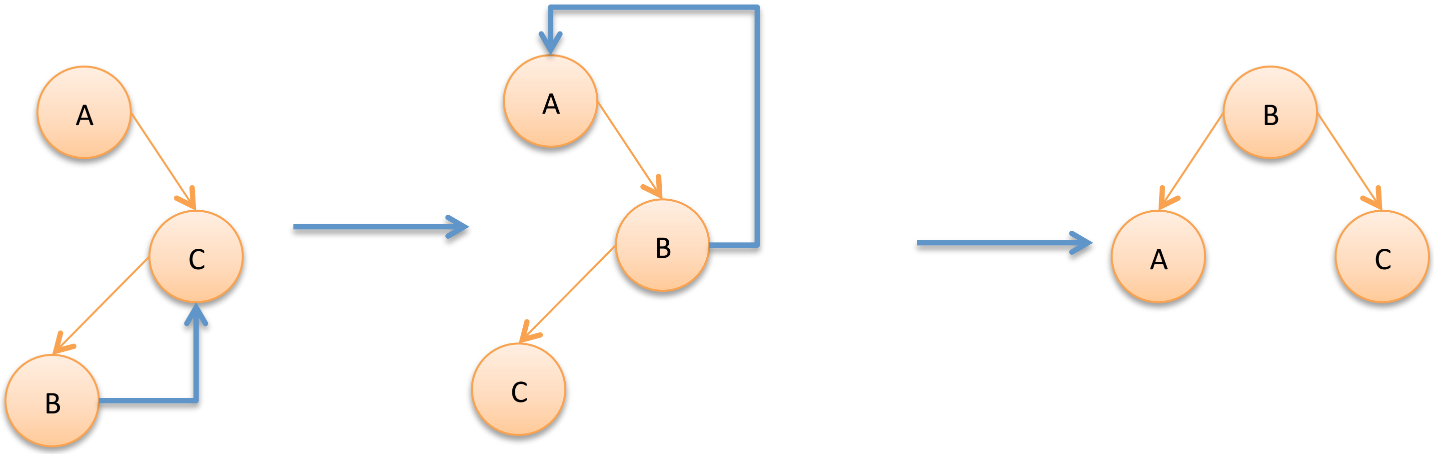



- Problem symmetric to the left-to-left insertion – not balanced anymore
- To correct this insertion – right rotation:
  - B must become the root
  - A must become the right child of B

# General Rotation for Right-to-Right Insertion

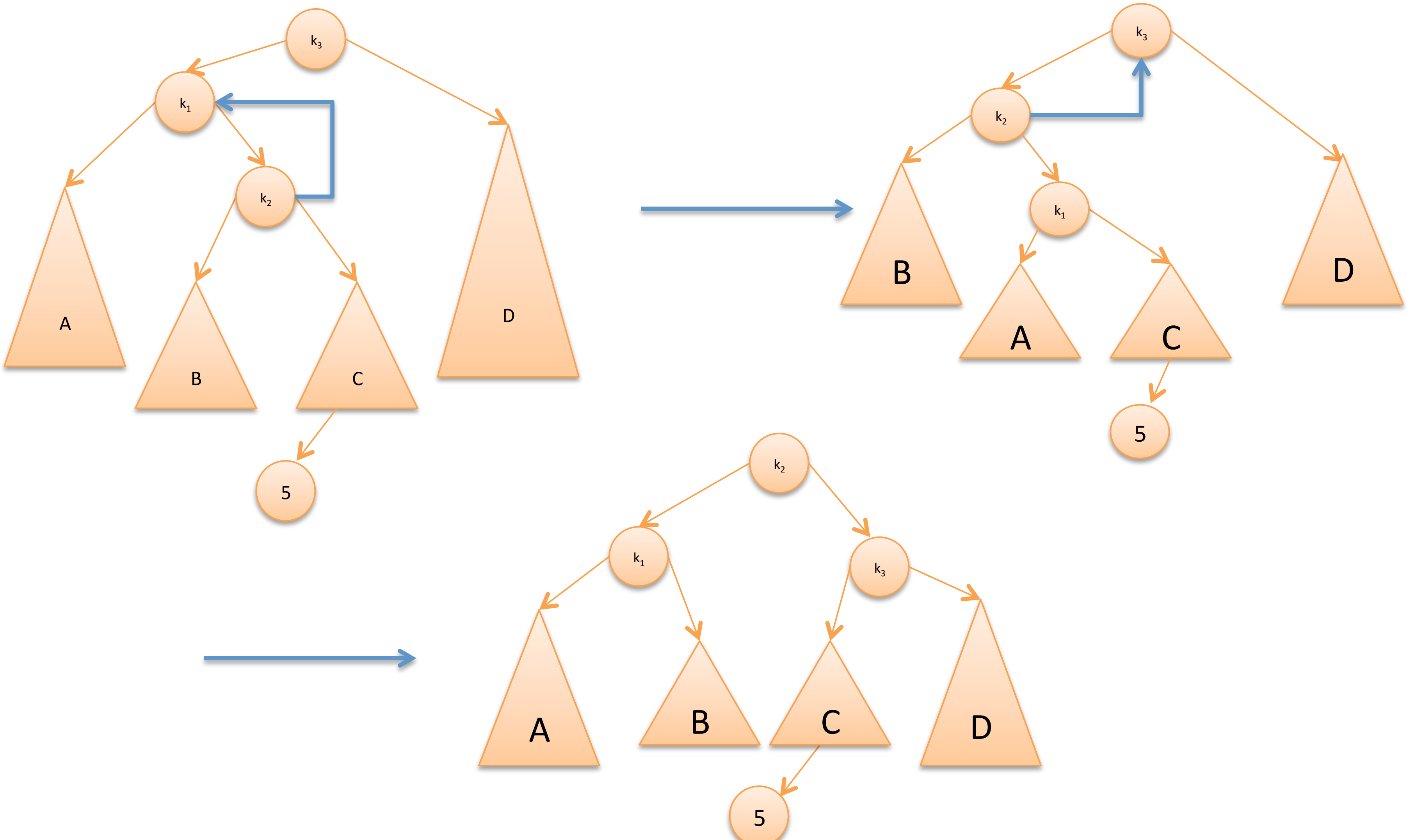


# Left-Right Insertion and Double AVL Rotation

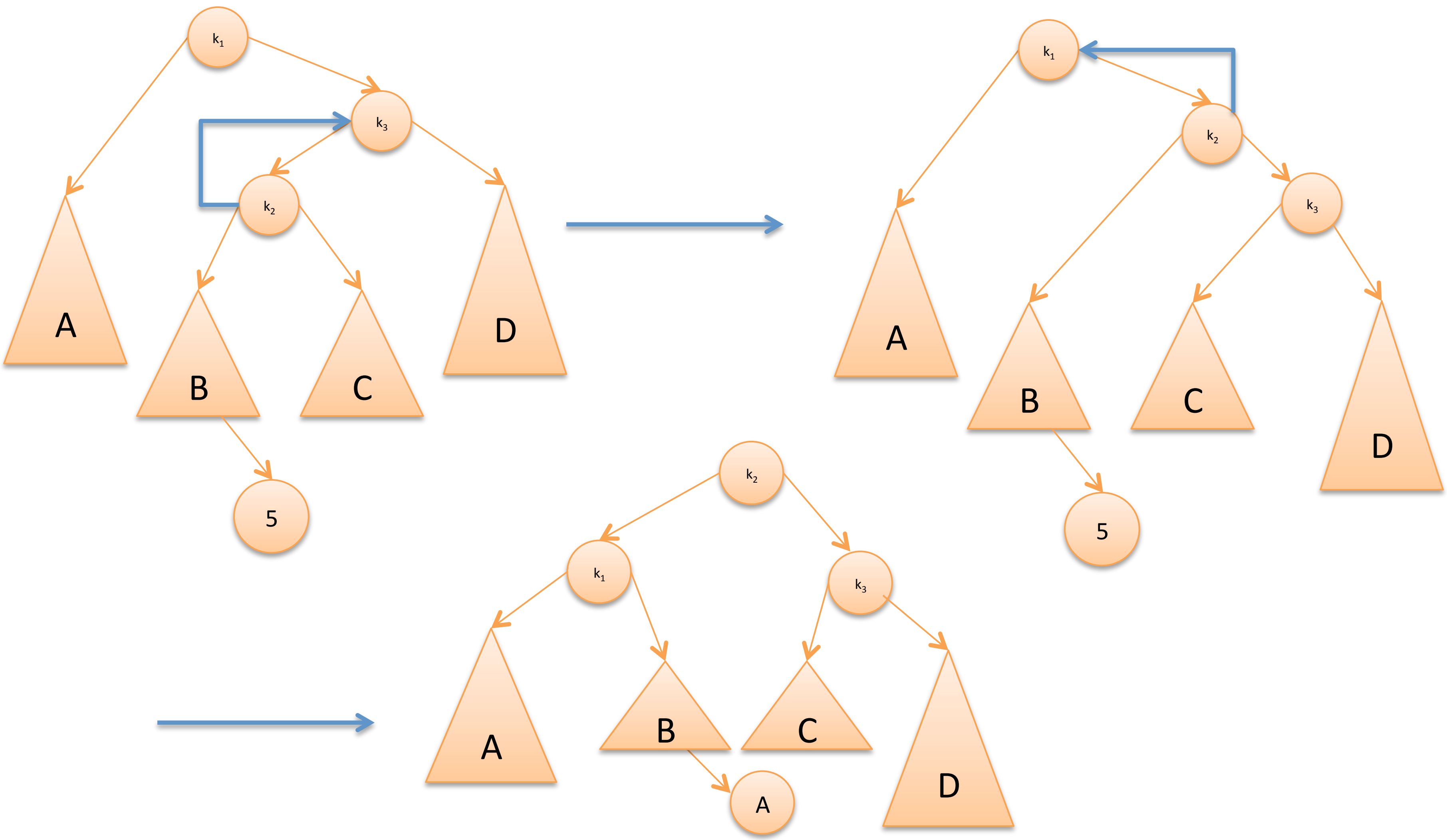



- Problem: not balanced anymore
- Identifying what should be the new root is key
- Imagine "lifting" up the root
- Where will the children have to go to maintain the search property?

# General Double Rotation for Left-Right Insertion



# General Double Rotation for Right-Left Insertion



# Class AVLNode

```
private static class AvlNode<T> {
T element; //data in the node
AvlNode<T> left; //Left child
AvtlNode<R> right; //Right child
int height; //Height


// Constructors
AvlNode(T theElement) {
    this(theElement, null, null);
}

AvlNode(T theElement, AvlNode<T> left, AvlNode<T> right) {
    element = theElement;
    left = left;
    right = rightt;
    height = 0;
}
```

# Method insert

```
/**
  * Internal method to insert into a subtree.
  * @param x the item to insert.
  * @param t the node that roots the subtree.
  * @return the new root of the subtree.
 */

private AvlNode<T> insert (T x, AvlNode<T> t) {

    if (t == null)
        return new AvlNode(x, null, null);
    int compareResult = x.compareTo(t.element);

    if(compareResult < 0)
        t.left = insert(x, t.left);
    else if(compareResult > 0)
        t.right = insert(x, t.right);
    else
        return balance(t); //Duplicate, do nothing
}
```

# Method balance

```
private static final int ALLOWED_IMBALNCE = 1;

//Assume t is either balanced or within one of being balanced
Private AvlNode<T> balance(AvlNode<T> t) {
    if(t == null)
        return t;
    if(t.left.height - t.right.height > ALLOWED IMBALANCE)
        if(t.left.left.height >= t.left.right.height)
            t.rotateWithLeftChild(t);
        else
            t.doubleWithLeftChild(t);
    else(t.right.height - t.left.height > ALLOWED IMBALANCE)
        if(t.right.right.height >= t.right.right.height)
            t.rotateWithRightChild(t);
        else
            t.doubleWithRightChild(t);

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

# Method `rotateWithLeftChild`

```
/**
*Rotate binary tree node with left child.
*For AVL tree, this is a single rotation for case 1.
*Update heights, then return new root.
*/

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

# Method `doubleWithLeftChild`

```
/**
*Double rotate binary tree, first left child with its right child,
then node k3 with new left child.
*For AVL tree, this is a double rotation for case 2.
*Update heights, then return new root.
*/

private AvlNode<T> doubleWithLeftChild(AvlNode<T> k3) {
    k3.left = rotateWithrightChild(k3.left);
    return rotateWithLeftChild(k3);
    }
```

# AVL Trees – Concluding Remarks

- If AVL rotation can enforce O(log n) height, what are the asymptotic runtimes for our functions?
  - insert(T x, AvlNode node) = O(log n) + balancing
  - find(T x) : O(height) = O(log n)

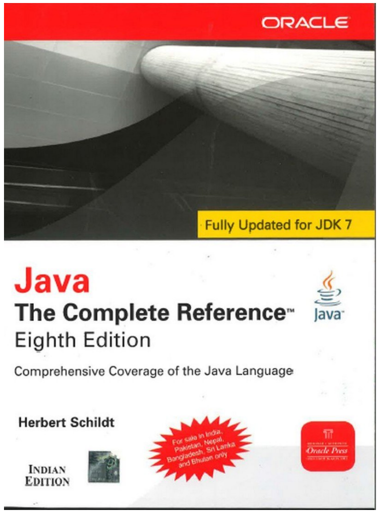
- How expensive is balancing?
- There are at most three nodes and four subtrees to move around → O(1)

Algorithms and Data Structures 2

# SETS AND MAPS

# Sets

- Write a program to count the number of occurrences of every unique word in a large text file (e.g. *Java Reference Manual*)




[Pictures credit: https://images-na.ssl-images-amazon.com/images/I/61rKnDmww9L.jpg]

- Possible approach: sets

# Sets

- Write a program to count the number of occurrences of every unique word in a large text file (e.g. *Java Reference Manual*)

- Possible approach:
  - Store the words in a collection
  - Report the # of unique words
  - Additionally: once you have this collection, allow auser to search it, to see whether various words appear in the text file

- Question: what is an appropriate data collections for this?
- Answer: Set

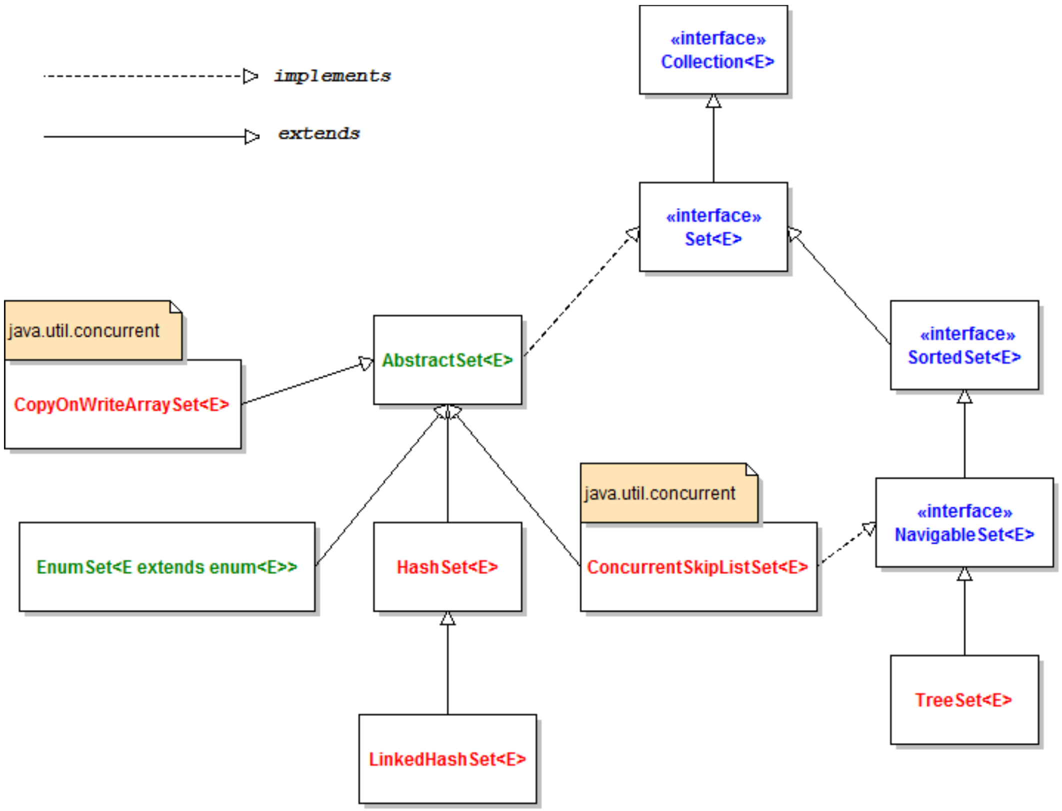
# Sets


[Pictures credit: https://www.beyondtheblackboard.com/components/com_virtuemart/shop_image/product/full/SET---Box---Transparent-Background---8-22-11_0.png]

# Sets

- **Set** - a collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - `add`,
  - `remove`,
  - `search (contains)`
- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order

# Set API Class Diagram




[Pictures credit: http://www.codejava.net/images/articles/javacore/collections/Set%20API%20class%20diagram.png]

# `Set` Implementations

- In Java, sets are represented by `Set` type in `java.util`
- Set is implemented by `HashSet` and `TreeSet` classes
  - `HashSet:` implemented using a "hash table" array
    - Very fast: O(1) for all operations
    - Elements are stored in unpredictable order
  - `TreeSet`: implemented using a binary search tree
    - Pretty fast: O(log N) for all operations
    - Elements are stored in sorted order
  - `LinkedHashSet:`
    - O(1) but stores in order of insertion, but slightly slower than `HashSet` because of extra info stored

# `Set` Methods

- We can construct an empty set, or one based on a given collection

- Examples:

```
Set<Integer> set = new TreeSet<Integer>(); // empty

List<String> list = new ArrayList<String>();
...
Set<String> set2 = new HashSet<String>(list);
```

# `Set` Methods

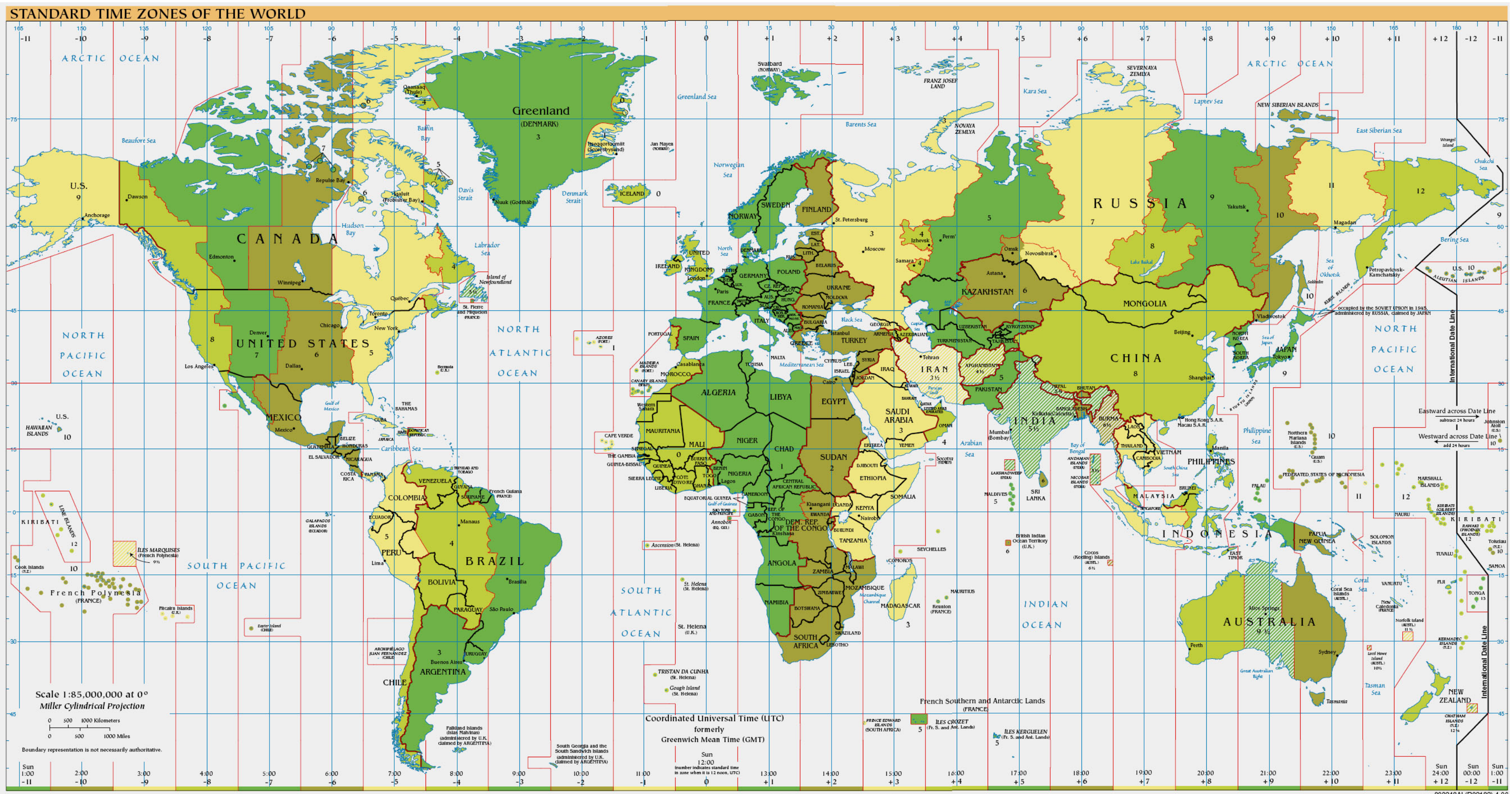
| | |
|---|---|
| **`add(value)`** | adds the given value to the set |
| `contains(`**`value`**`)` | returns true if the given value is found in this set |
| `remove(`**`value`**`)` | removes the given value from the set |
| `clear()` | removes all elements of the set |
| `size()` | returns the number of elements in list |
| `isEmpty()` | returns true if the set's size is 0 |
| `toString()` | returns a string such as "[3, 42, -7, 15]" |

# Maps

- Write a program that stores, modifies and retrieves:
  - Assignment grades for every student in this College
  - Financial information for every client of some bank
  - Browsing history for every user of some search engine
  - Searches and transactions for every user of some online retailer
  - Activity and likes of every user of some online platform

- Question: What do these records have in common?
- The way we think about them → every data sample has a unique user → unique ID (key)
- What is the appropriate data collection for this data?
- Maps

# Maps




[Pictures credit: https://theodora.com/maps/new9/time_zones_4.jpg]

# Maps

- **Map** – a data collection that holds a set of unique *keys* and a collection of *values*, where each key is associated with one value
- Also known as:
  - Dictionary
  - Associative array
  - Hash
- Basic map operations:
  - **`put`**`(`*`key`*`, `*`value`*`)` - adds a mapping from a key to a value
  - **`get`**`(`*`key`*`)` - retrieves the value mapped to the key
  - **`remove`**`(`*`key`*`)` - removes the given key and its mapped value

# Map Implementations

- In Java, maps are represented by Map type in java.util
- Map is implemented by the HashMap and TreeMap classes
  - HashMap - implemented using a "hash table"
    - Extremely fast: **O(1)**
    - Keys are stored in unpredictable order
  - TreeMap - implemented as a linked "binary tree" structure
    - Very fast: **O(log N)**
    - Keys are stored in sorted order
  - LinkedHashMap - O(1)
    - Keys are stored in order of insertion

# `Map` Implementations

- Map requires 2 types of parameters:
  - One for keys
  - One for values

- Example:

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```

# Map Methods

| | |
|---|---|
| **put(key, value)** | adds a mapping from the given key to the given value;<br>if the key already exists, replaces its value with the given one |
| get(**key**) | returns the value mapped to the given key (null if not found) |
| containsKey(**key**) | returns true if the map contains a mapping for the given key |
| remove(**key**) | removes any existing mapping for the given key |
| clear() | removes all key/value pairs from the map |
| size() | returns the number of key/value pairs in the map |
| isEmpty() | returns true if the map's size is 0 |
| toString() | returns a string such as "{a=90, d=60, c=70}" |

# `keySet` and `Values`

- `keySet` method returns a **`Set`** of all keys in the map
  - It can loop over the keys in a `foreach` loop
  - It can get each key's associated value by calling `get` on the map
- Example:

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2); // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) {        // Geneva -> 2
    int age = ages.get(name);                    // Marty -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

# Methods `keySet` and `Values`

- `values` method returns a collection of all values in the map
  - It can loop over the values in a `foreach` loop
  - No easy way to get from a value to its associated key(s)

| | |
|---|---|
| `keySet()` | returns a set of all keys in the map |
| `values()` | returns a collection of all values in the map |
| `putAll(`**map**`)` | adds all key/value pairs from the given map to this map |
| `equals(`**map**`)` | returns `true` if given map has the same mappings as this one |

# Example: Maps

- Many words are similar to other words
- For example word wine can become:
    - dine, fine, line, mine, nine, pine, or vine
    - wide, wife, wipe, or wire
    - wind, wing, wink, or wins
- Write a program to find all words that can be changed into at least 15 other words by a single one-character substitution
- Assume that:
    - We have a dictionary consisting of approximately 89,000 different words of varying lengths
    - Most words are between 6 and 11 characters

[Coding Example]

Algorithms and Data Structures 2

# HASHING AND HASH FUNCTIONS

# Introduction to Hashing

- Suppose we have a large set of items *M*
- Set *M* contains the subset *D*, where *D* << *M* of items that we actually care about
- Example: *M* may be English dictionary, and *D* maybe a set of English words we use in everyday life
- Problem: How to store data such that we use only O(*D*) memory, while achieving fast O(1) access

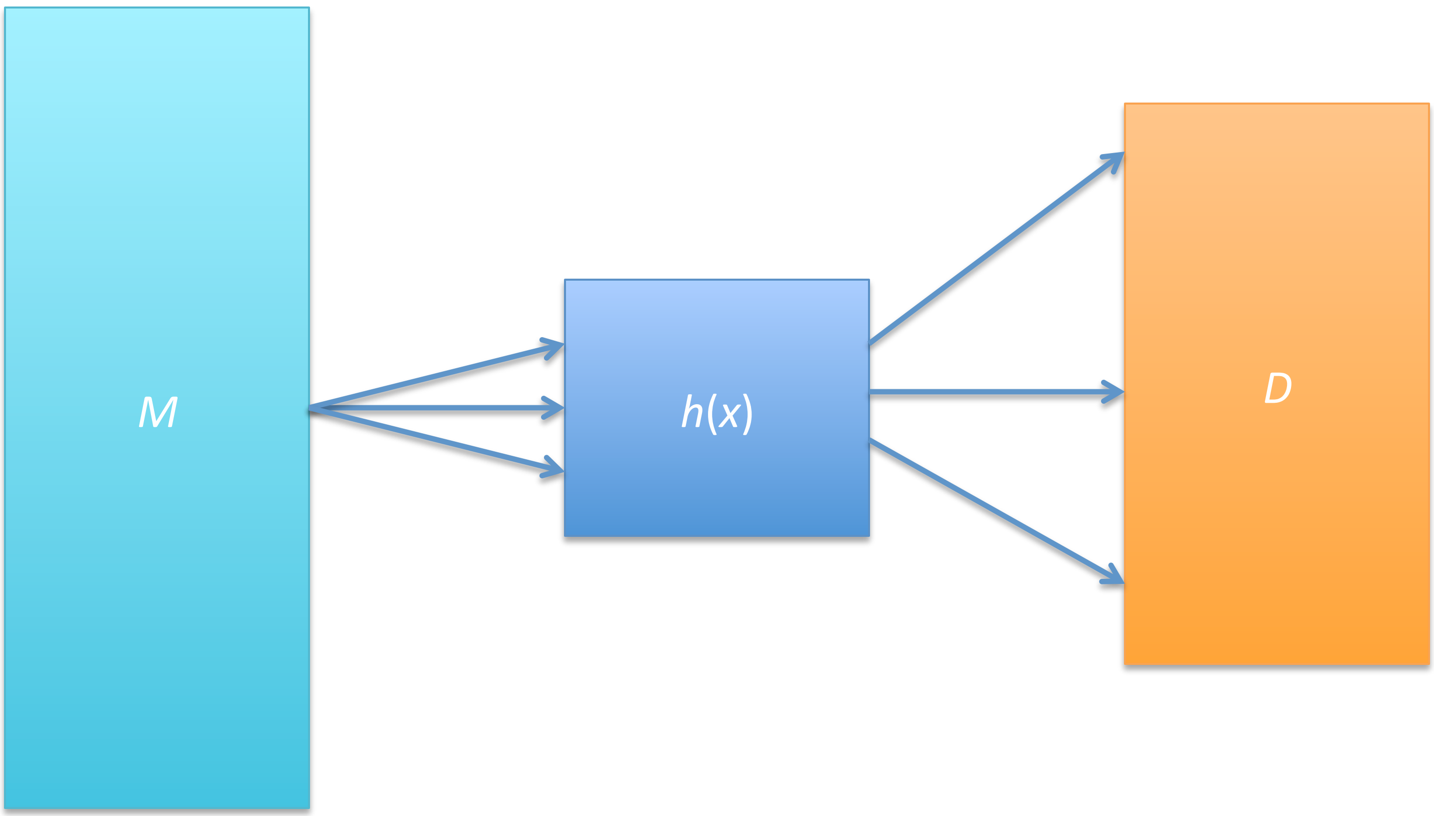
# Introduction to Hashing

- Memory: A Hash Table
  - Consider an array of size **c** * *D*
  - Each index in the array corresponds to *some* element in *M* that we want to store
  - The data in **D** does not need any particular ordering

- Possible approaches:
  - Unsorted array – search takes O(*D*)
  - Sorted array – search still takes O(*D*)
  - Random mapping – search still takes O(*D*)

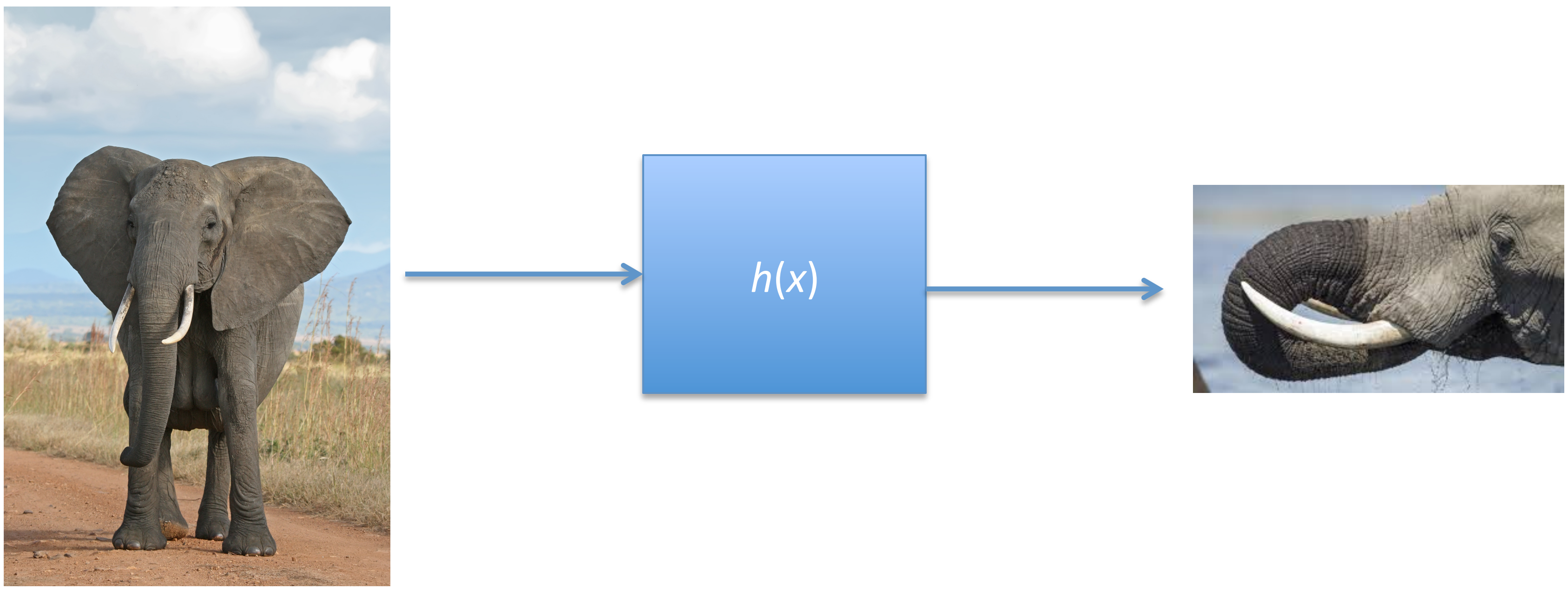
# Introduction to Hashing

- Another possible approach – pseudo-random mapping using a hash function $h(x)$



# Introduction to Hash Functions

- Hash function - maps the large space M into target space D




[Pictures credit: https://upload.wikimedia.org/wikipedia/commons/3/37/African_Bush_Elephant.jpg
https://aos.iacpublishinglabs.com/question/aq/1400px-788px/what-are-elephant-tusks-used-for_3c174bec-bd85-4fab-a617-7a1a33f14c62.jpg?domain=cx.aos.ask.com]

# Introduction to Hash Functions

- **Hash function** - maps the large space M into target space D

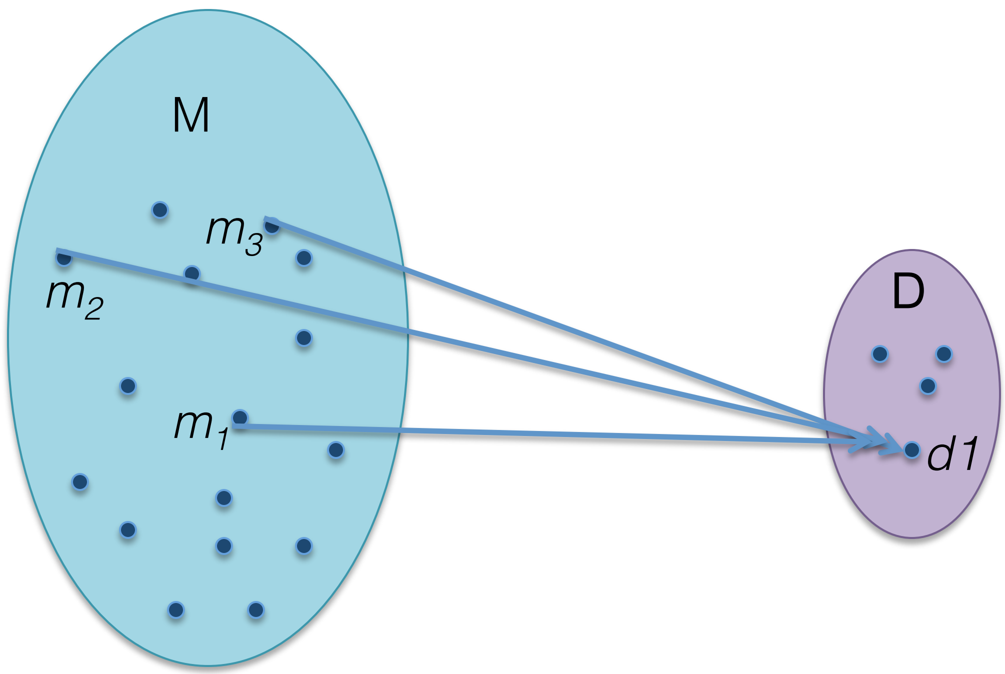
- Desired properties of hash functions:
  - Repeatability:
    - For every $x$ in $D$, it should always be h($x$) = h($x$)
  - Equally distributed:
    - For some $y$, $z$ in D, P(h(y)) = P(h(z))
  - Constant-time execution: h($x$) = O(1)

# Simple Hash Function

```
/**
* A hash method for String objects.
* @param key the String to hash.
* @param tableSize the size of the hash table.
• @return the hash value.
  */
  public static int hash(String key, int tableSize) {
      int hashVal = 0;

      for(inti=0; i<key.length();i++)
         hashVal = 37 * hashVal + key.charAt(i);

      hashVal %= tableSize;
      if(hashVal < 0)
         hashVal += tableSize;
      return hashVal;
}
```

# Problems with Hash Functions

- Hash function – can be thought of as a "lossy compression function", since |M| << |D|




- Do you see any problems here?
- Yes, collision!

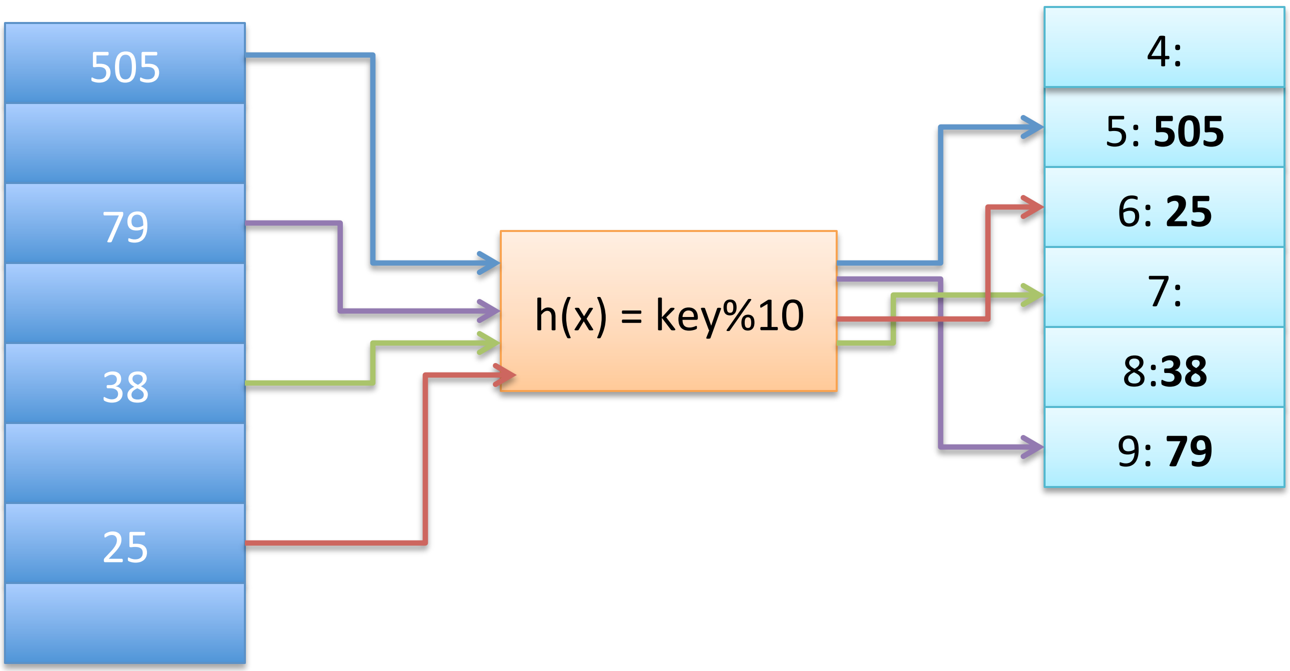
# Resolving Collisions

- Hash function – can be thought of as a "lossy compression function", since |M| << |D|
- Problem – collision

- Possible approaches to resolve collisions:
  - Store data in the next available space
  - Store both in the same space
  - Try a different hash
  - Resize the array

# Resolving Collisions – Linear Probing

- Linear probing – a simple approach
  - When a collision occurs, find the next available spot in the array



# Problems with Linear Probing

- Linear probing – a simple approach
  - When a collision occurs, find the next available spot in the array
- Searching for some element *x*:
  - Go to position *h*(*x*), then cycle through all entries until you either find the element, or the blank space
- Adding an element *y*, that should go to the position taken by the colliding element:
  - Add element *y* to the next available spot - clustering

# Problems with Linear Probing

- If a cluster becomes too large, it may have negative consequences on the hashing performance:
  - The chances of collision with the cluster increase
  - The time it takes to find something in the cluster increases, and it **isn't O(1)**

# Quadratic Probing

- Whereas linear probing increments indices by one each time, quadratic probing goes through the squares

- For example, linear probing would check index 3, then:
  - 3+1,
  - 3+2,
  - 3+3,
  - 3+4 etc.
- Quadratic probing would check index 3, then
  - 3+1,
  - 3+4
  - 3+9
  - 3+16 etc.

# Problems with Quadratic Probing

- Example: Consider a hash function for ints, h(x) = x%7
- Insert, 3, 10,17,24,31,38
- What happens? Where does 31 go?
- 31%7=3
- 3+1%7 = 4
- 3+2%7 = 5
- 3+4%7 = 0
- 3+9%7 = 5
- 3+16%7 = 5

| |
|---|
| 0: 17 |
| 1 |
| 2 |
| 3: 3 |
| 4:10 |
| 5: 24 |
| 6 |

# Problems with Quadratic Probing

- Secondary clustering problem
  - Even when there is space available in the table, quadratic probing is not guaranteed to find an opening
  - In fact, half the array has to be empty to guarantee an opening
  - This approach reduces the O(n) problem of linear probing, but it introduces even larger memory constraints
  - .

# Secondary Hashing

- If two keys collide in the hash table, then a secondary hash indicates the probing size
- Need to be careful, possible for infinite loops with a very empty array

# Chaining

- Rather than probing for an open position, we could just save multiple objects in the same position
- Some data structure is necessary here
- Commonly a linked list, AVL tree or secondary hash table

Resizing isn't **necessary**, but if you don't, you will get O(n) runtime

# Hash Functions

- In reality, good hash functions are difficult to produce
- We want a hash that distributes our data evenly throughout the space
- Usually, our hash function returns some integer, which must then be moded to our table size
- When discussing hash table efficiency, we call the proportion of stored data to table size the *load factor*

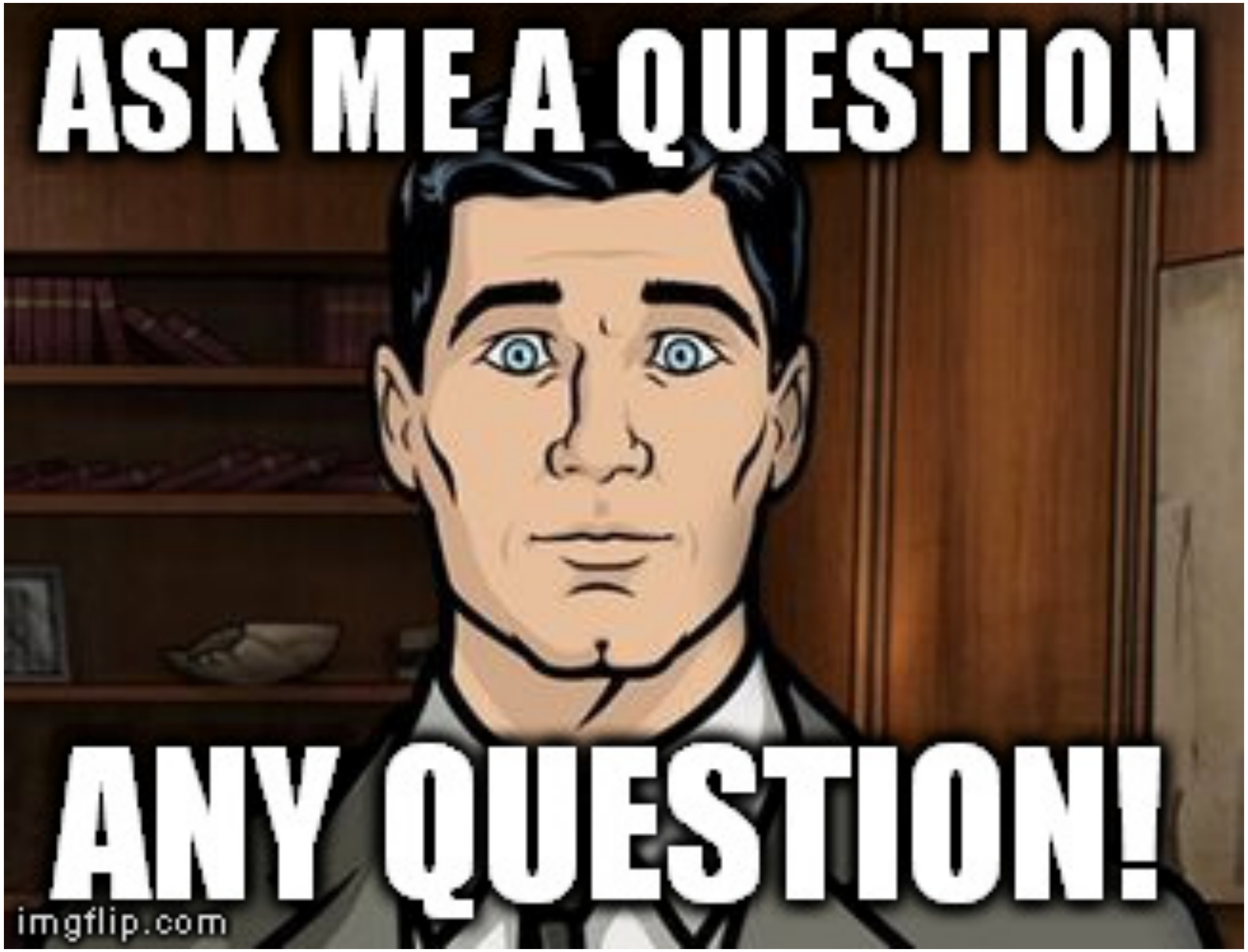
# Java HashTable Class

- Implements a hash table, which maps keys to values

- Example: a HashTable of integers

```
Hashtable<String, Integer> number = new
HashTable<String, Integer>();
numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);

Integer n = numbers.get("two");
if(n != null) {
   System.out.println("two = " +n);
}
```

# Your Questions




[Meme credit: imgflip.com]

# References and Reading Material


- Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, chapters 4, 5 and 9
- Oracle, java.util Class Collections, [Online] http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html
- Oracle, Java Tutorials Collections, [Online] https://docs.oracle.com/javase/tutorial/collections/
- Vogella, Java Collections – Tutorial, [Online] http://www.vogella.com/tutorials/JavaCollections/article.html
- TutorialsPoint, Data Structures and Algorithms - AVL Trees, [Online], https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm
- AVL Trees Animation, [Online], https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
- Sets, [Online], https://web.cs.wpi.edu/~cs2102/common/kathi-notes/sets.html
- Java Tutorials, Sets, [Online], https://docs.oracle.com/javase/7/docs/api/java/util/Set.html
- Java Tutorials, Interface Map, [Online], https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
- Problem Solving with Algorithms and Data Structures, Hashing, [Online], http://interactivepython.org/courselib/static/pythonds/SortSearch/Hashing.html (note: code in Python)