Recursive data types

Tutorial 7 (12th April 2022)

Ike Mulder

instanceof

Beware of **instanceof** operator

Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself [Scott Meyers]

```
public abstract class Animal {}

public class Cat extends Animal {
  public String meow() {
    return "meow, meow";
  }
}

public class Dog extends Animal {
  public String bark() {
    return "woof, woof";
  }
}
```

Don't!

```
public class BadInstanceOf {
  public static void makeSound(Animal a){

    if (animal instanceof Cat) {

      Cat cat = (Cat) a;
      System.out.println( cat.meow() );

    } else if (animal instanceof Dog) {
      Dog dog = (Dog) a;
      System.out.println( dog.bark() );
    }
  }
}
```

instanceof

- Use *polymorphism*

```java
public abstract class Animal {
  abstract String makeSound ();
}

public class Cat extends Animal {
  @Override
  public String makeSound() {
    return "meow, meow";
  }
}

public class Dog extends Animal {
  @Override
  public String makeSound() {
    return "woof, woof";
  }
}
```

```java
public class GoodPolymorphism {
  public static void makeSound(Animal a){
    System.out.println(a.makeSound());
  }
}
```

Expression assignment – partialEval, bad

```
public class Add extends TwoArgExpr {
    private Expr left, right;

    private Expr partialEval() {
       if (left instanceof Variable) {
           if (right instanceof Constant) {
               ...
           } else if (right instanceof Variable) {
               ...
           }
       }
    }
```

Expression assignment – partialEval, better

```
public class Add extends TwoArgExpr {
    private Expr left, right;

    private Expr partialEval() {
       Expr leftPartial = left.partialEval();
       Expr rightPartial = right.partialEval();
       double leftConst = leftPartial.getConstantValue();
       double rightConst = rightPartial.getConstantValue();
       if (leftConst != null && rightConst != null) {
           return Constant(leftConst + rightConst);
       }
       if (leftConst != null && leftConst == 0.0) {
           return rightPartial;
       }
       ...
    }
```

Recursive data types

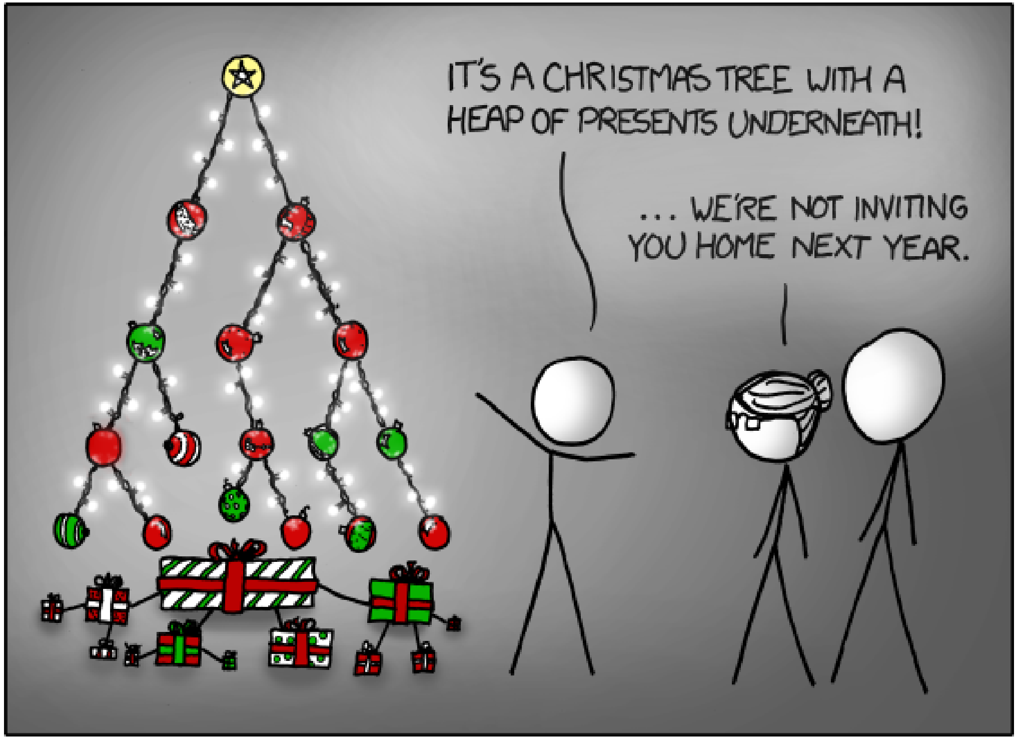
Traversal of recursive data types possible using

- loops
- iterators
- *recursion*

for all operations on recursive data structures, recursion is your friend:

- consider the base case
- consider the recursive case

Trees: more than one child (recursive reference)


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

Binary search tree

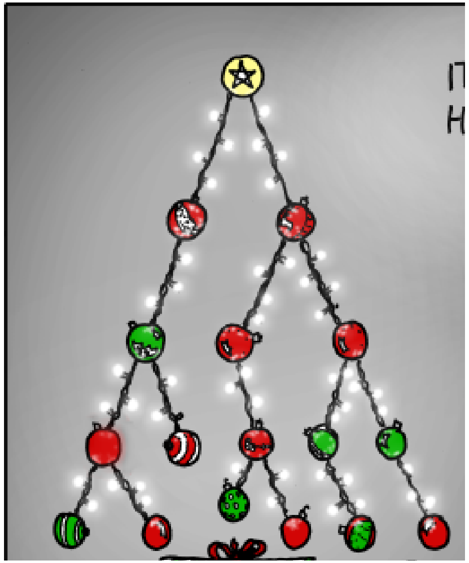
```
public class Tree <E extends Comparable<E>> {

    protected Node root;

    private class Node {
        private E el;
        private Node left, right;

        public Node( E e, Node l, Node r ) {
            el = e;
            left = l;
            right = r;
        }

        public Node( E e ) {
            this( e, null, null );
        }
    }
```

very similar to list, only with two children

Radboud University

Trees with different kinds of nodes

- Node is a class like any other,
  we can have subclasses for different variants



Trees with different kinds of nodes

```
public class Tree0_1_2 <E> {
    private Node root;

    private abstract class Node {
        private E e;
        public Node( E x ) {
            e = x;
        }
        public abstract int size();
    }
}
```

No successor: the leaves of the tree

```
private class Node0 extends Node {
    public Node0( E e) {
        super( e );
    }
    @Override
    public int size() {
        return 1;
    }
}
```

no successors

One successor

```
private class Node1 extends Node {
    private Node next;
    public Node1( E e, Node n ) {
        super( e );
        next = n;
    }
    public Node1( E e ) {
        this( e, null );
    }
    @Override
    public int size() {
      return ( next == null ? 0 : next.size()) + 1 ;
    }
}
```

Two successors

```
private class Node2 extends Node {
    private Node left, right;
    public Node2( E e, Node l, Node r ) {
        super(e);
        left  = l;
        right = r;
    }

    public Node2( E e ) {
        this( e, null, null);
    }

    @Override
    public int size() {
        return (left  == null ? 0 : left.size())  +
               (right == null ? 0 : right.size()) + 1;
    }
}
```

Counting the nodes in a tree

```
public class Tree0_1_2 <E> {
    private Node root;
        ...
    public int size() {
        return root == null ? 0 : root.size();
    }
```

- the method implementations belong to the subtypes
- dynamic binding guarantees that the right implementation of size will be called.

a single recursive method

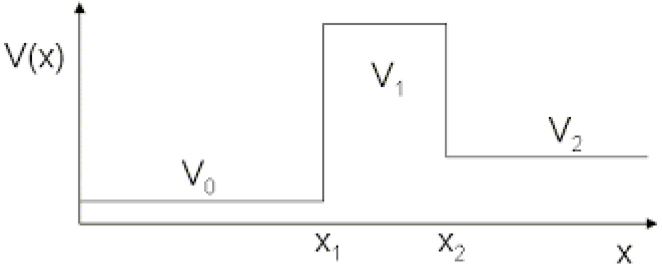
```
public static int size( Node n ) {
    if ( n == null ) {
        return 0;
    } else if ( n instanceof Node0 ) {
        return 1;
    } else if ( n instanceof Node1 ) {
        Node1 n1 = (Node1) n;
        return 1 + size( n1.next );
    } else if ( n instanceof Node2 ) {
        Node2 n2 = (Node2) n;
        return 1 + size( n2.left ) + size( n2.right );
    } else {
        throw new IllegalArgumentException();
    }
}
```

dynamic binding is better

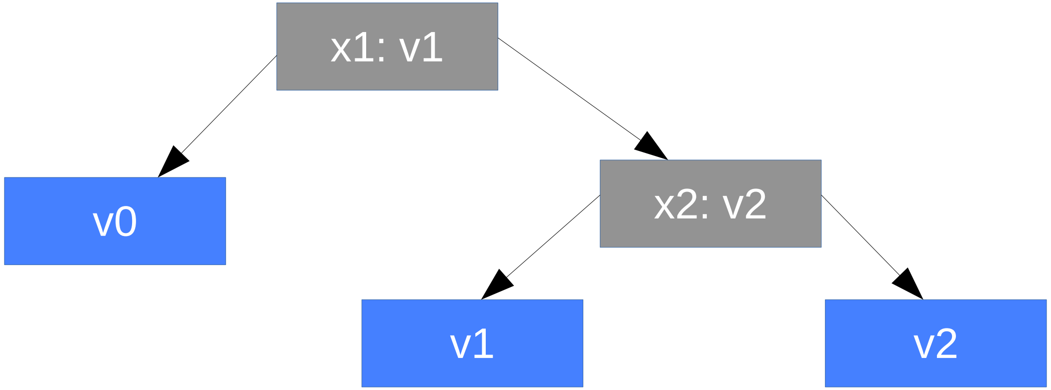
needed type casts are ugly

IntervalTreeMap

- Want to model functions like:



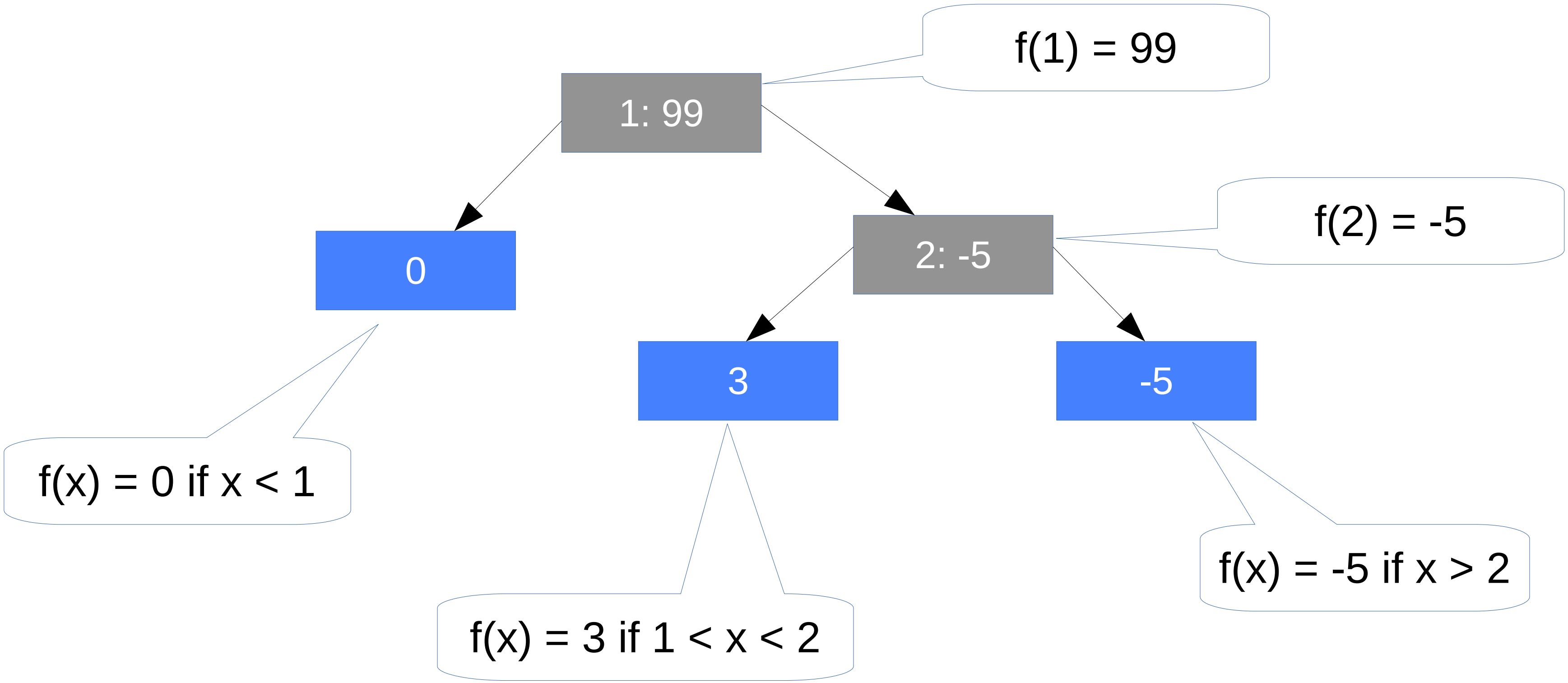
- Idea: represent as binary tree, but leaves have a value



Leaves now carry data!

Can no longer use null

IntervalTreeMap - getValue


1: 99

f(1) = 99

0

f(x) = 0 if x < 1

2: -5

f(2) = -5

3

f(x) = 3 if 1 < x < 2

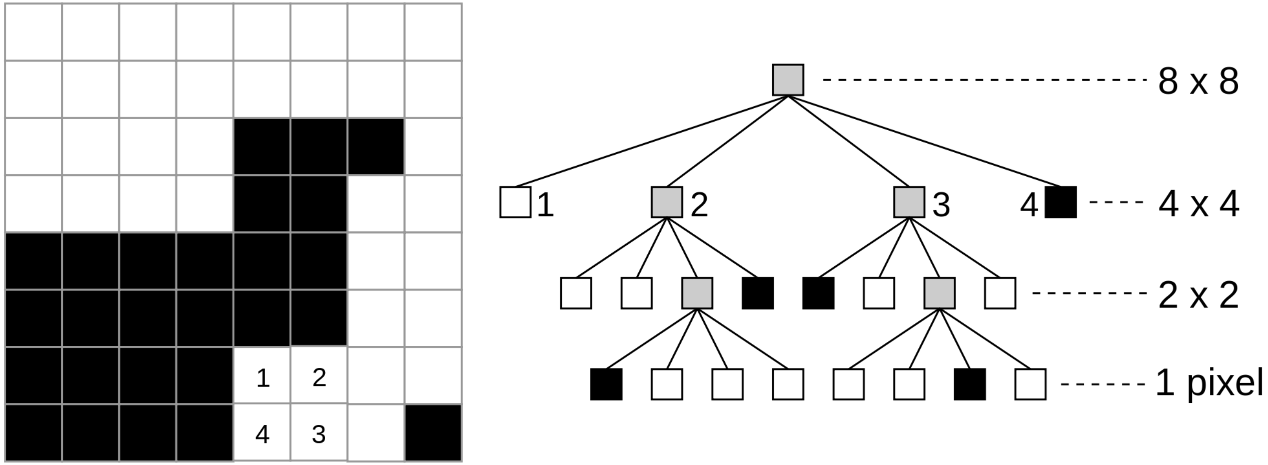
-5

f(x) = -5 if x > 2

IntervalTreeMap - demo

Quadtrees

- image compression, collision detection
- idea:
  1. A (sub)image that is entirely white or black is represented by a single white or black node, respectively.
  2. Otherwise the image is divided into 4 subimages. Each subimage is represented recursively as a quadtree. These 4 quadtrees are combined using an internal (grey) node.



QuadTree design

- top-level QuadTreeNode: interface
- subtype for each different node type
  - white nodes: WhiteLeaf
    - square is entirely white
  - black nodes: BlackLeaf
    - square is entirely black
  - grey node: GreyNode
    - always 4 subtrees, with different colors
- operations become recursive methods
  - define operations as methods of the interface
  - make an implementation in each subclass

0	1
3	2

example: compute number of black pixels (I)

```
public interface QTNode {
    public int countBlackPixels( int size );
}
public class WhiteLeaf implements QTNode {
    @Override
    public int countBlackPixels( int size ) {
        return 0;
    }
}
public class BlackLeaf implements QTNode {
    @Override
    public int countBlackPixels( int size ) {
        return size * size;
    }
}
```

example: compute number of black pixels (II)

```
public class GreyNode implements QTNode {
    private final QTNode[] children;

    @Override
    public int countBlackPixels( int size ) {
        int blacks = 0;
        for ( QTNode node: children )
            blacks += node.countBlackPixels( size  / 2 );
        return blacks;
    }
}
```

Alternatively: leaves as enum

```java
public enum Leaf implements QTNode {
    Black( false ), White( true );

    private final boolean isWhite;

    private Leaf( boolean isWhite ) {
        this.isWhite = isWhite;
    }

    @Override
    public int countBlackPixels( int size ) {
        return isWhite ? 0 : size * size ;
    }
}
```

Finally