Recursive data types: Quadtrees

Tutorial 6 (3rd March 2021)

Ike Mulder

Radboud University

`instanceof` and **`null`**

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";
  }
}
```


```
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*

```
public class Animal {
  public String makeSound () {
    return "<silence>";
  }
}

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

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

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

`null` or The worst mistake of computer science

I call it my billion-dollar mistake… At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
[Tony Hoare, 2009]

- What is wrong with **`null`?**
  - **`null`** is a value that is not a value.

null

- Example: (tempting) **null** used to indicate the absence of a value

```
public class Person {
  private Car car;
  public Car getCar() {
    return car;
  }
}
```

```
public class Car {
  private Insurance insurance;
  public Insurance getInsurance() {
    return insurance;
  }
}
```

```
public class Insurance {
    private String name;
    public String getName() {
        return name;
    }
}
```

```
String getCa    suranceName(Person person){
  return pers    getCar()
                   Insurance()
               .get   me();
}
```

```
String getCarInsuranceName(
   Person person
){
  if (p
    Car
    if
      I
                                 e();

      i
                                 () }

      }
  }
  return "Unknown";
}
```

null

- Example: (tempting) **null** used to indicate the absence of a value

```
public class Person {
  private Car car;
  public Car getCar() {
    return car;
  }
}
```

```
public class Car {
  private Insurance insurance;
  public Insurance getInsurance() {
    return insurance;
  }
}
```

```
public class Insurance {
    private String name;
    public String getName() {
        return name;
    }
}
```

```
String getCarInsuranceName(
   Person person
) {
  if (person == null) {
    return "Unknown";
  }
  Car car = person.getCar();
  if (car == null) {
    return "Unknown";
  }
  Insurance insurance =
                  car.getInsurance();
  if (insurance == null) {
    return "Unknown";
  }
  return insurance.getName();
}
```

Exceptions i.s.o. null?

```java
public class NotPresentException extends Exception {
  public NotPresentException( String type ) {
    super (type + ": Element not present");
  }
}
```

```java
public class Person {
  private Car car;
  public Car getCar() throws NotPresentException {
    if ( car == null ) {
      throw new NotPresentException( "Person" );
    }
    return car;
  }
}
```

```java
String getCarInsuranceName( Person person ) {
  try {
    return person.getCar().getInsurance().getName();
  } catch (ElementNotPresentException ex) {
    return "<<Unknown>>";
  }
}
```

no!?

Exceptions i.s.o. `null`?

Java: Look Before You Leap (LBYL)

Python: Easier to Ask Forgiveness than Permission (EAFP)

- try catch is somewhat expensive
- Java is statically typed: we know more about the objects we’re dealing with

Exceptions i.s.o. `null`?

- Exceptions should be for Exceptional cases. These are unexpected scenarios, and usually, will not have a nice easy way of recovery.
- If there is a sensible recovery option, then do not use an exception.
- Using exceptions to control the flow through your program can have unintended side-effects
-

```
public static int readIntWithPrompt( String prompt ) {
    Scanner in = new Scanner( System.in );
    System.out.print(prompt);
    while( !in.hasNextInt() ) {
        in.nextLine();
        System.out.print(prompt);
    }
    return in.nextInt();
}
```

Optional

- We need something that contains information about
  (1) whether it holds a value, and
  (2) the contained value, if it exists.
- Such a ‘thing’ exists in Java: **`class`** `Optional<T>`

```
public class Person {
    private Car car;
    public Optional<Car> getCar() {
        return Optional.ofNullable(car);
    }
}
```

Creates an Optional that is empty if car == null, and contains the car otherwise

Optional

- What about getCarInsuranceName?

```
String getCarInsuranceName( Person person ) {
    if (person == null) {
       return "Unknown";
    }
    Car car = person.getCar();
    if (car == null) {
        return "Unknown";
    }
    Insurance insurance = car.getInsurance();
```

Becomes:

```
public String getCarInsuranceName( Optional<Person> person ) {
    return person.flatMap( (Person p) -> p.getCar() )
                 .flatMap( (Car c) -> c.getInsurance() )
                 .map( (Insurance ins) -> ins.getName() )
                 .orElse( "Unknown" );
}
```

Lambda expressions

methods of class Optional

Optional

- Becomes:

```
public String getCarInsuranceName( Optional<Person> person ) {
    return person.flatMap( (Person p) -> p.getCar() )
                 .flatMap( (Car c) -> c.getInsurance() )
                 .map( (Insurance ins) -> ins.getName() )
                 .orElse( "Unknown" );
}
```

methods of class Optional

Lambda expressions

```
flatMap: method of Optional<Person>
   given a function to get an Optional<Car> from a Person
   returns an Optional<Car>:
   - applies the function if there is some Person
   - returns Optional.empty() otherwise
```

Expressions (assignment 5)

```
public interface Expr {
   ...
   public default Optional<Double>
            toValue() {
      return Optional.empty();
   }
}
```

```
public class Constant implements Expr {
   ...
   @Override
   public Optional<Double> toValue() {
       return Optional.of( myValue );
   }
}
```

```
public class Add extends TwoArgExpr {
    ...

    private static BaseExpr symbolicAdd( BaseExpr x, BaseExpr y ) {
        Optional<Double> optX = x.toValue();
        Optional<Double> optY = y.toValue();
        if ( optX.isPresent() && optY.isPresent() ) {
            return new Constant( optX.get() + optY.get() );
        } else if ( optX.isPresent() && optX.get() == 0.0 ) {
            return y;
        } else if ( optY.isPresent() && optY.get() == 0.0 ) {
            return x;
        } else {
            return new Add(x, y);
        }
    }
```

Recursive data types

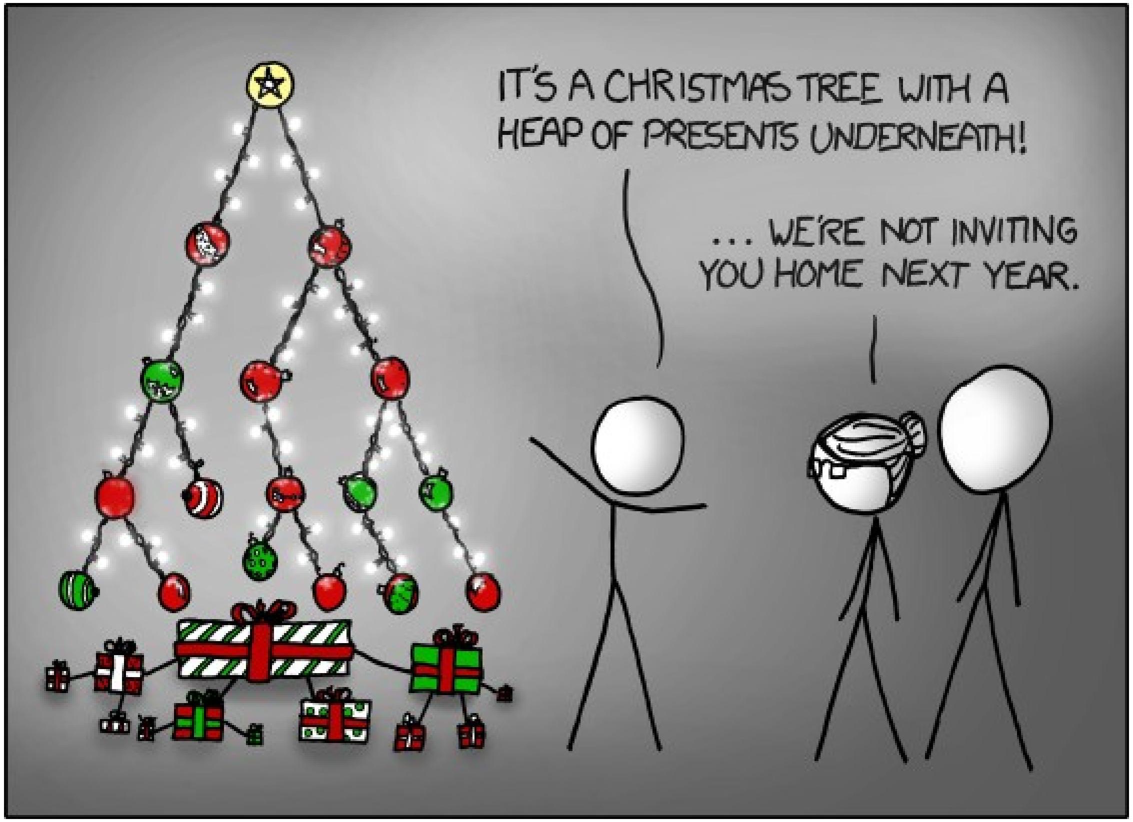
List traversal possible using

- 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)

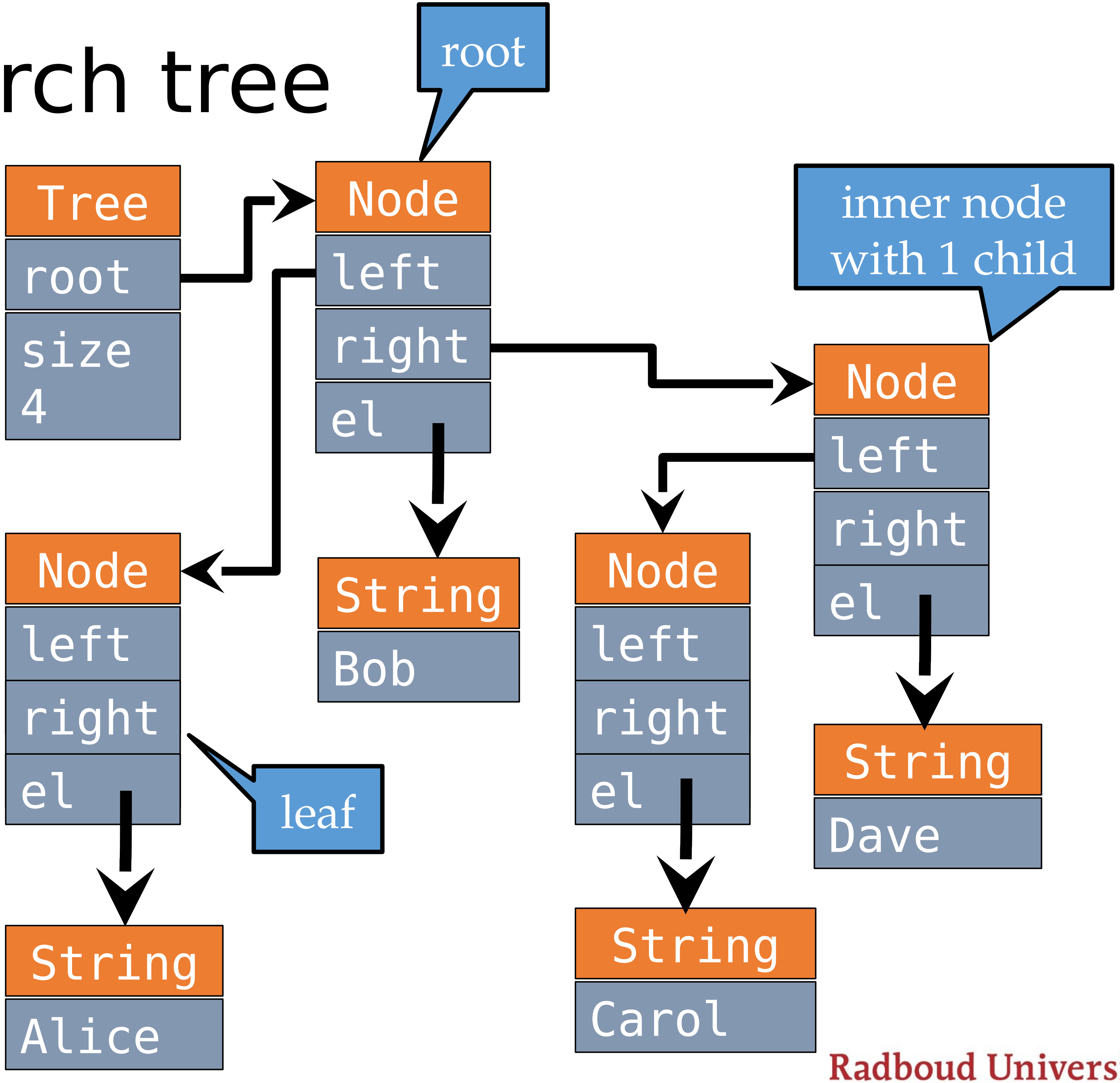


Trees

- Lists: nodes have one child
- Trees: nodes have two or more children
  - often we have different kind of nodes
    e.g. Leaf (no children) and Fork (with children)

- Binary trees are most common
  - (at most) two children
  - all elements in the left subtree are smaller than element in node
  - all elements in right subtree are greater
  - hence we allow no duplicates

search tree


root
Tree
root
size
4
Node
left
right
el
inner node
with 1 child
Node
left
right
el
String
Bob
Node
left
right
el
leaf
Node
left
right
el
String
Dave
String
Alice
String
Carol

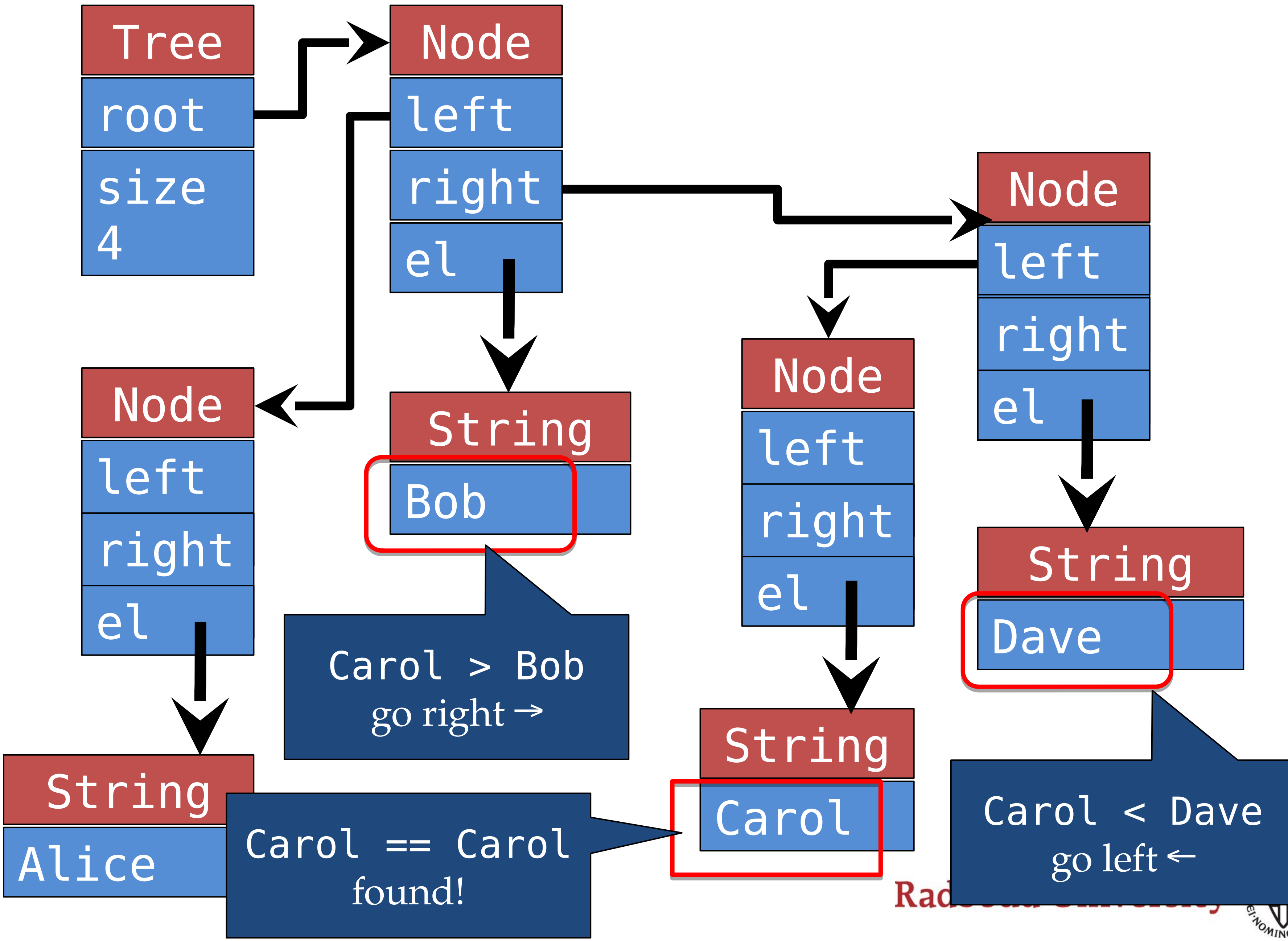
binary search tree

to provide element comparison

```
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

tree: does it contain Carol ?



contains in search tree

```
public boolean contains( E e ) {
    return contains( root, e );
}

private boolean contains( Node n, E e ) {
    if ( n == null ) {
        return false;
    } else {
        int comp = e.compareTo( n.el );
        if (comp < 0) {
            return contains( n.left, e );
        } else if (comp == 0) {
            return true;
        } else { // comp > 0
            return contains( n.right, e );
        }
    }
}
```

common pattern: helper method with reference to tree

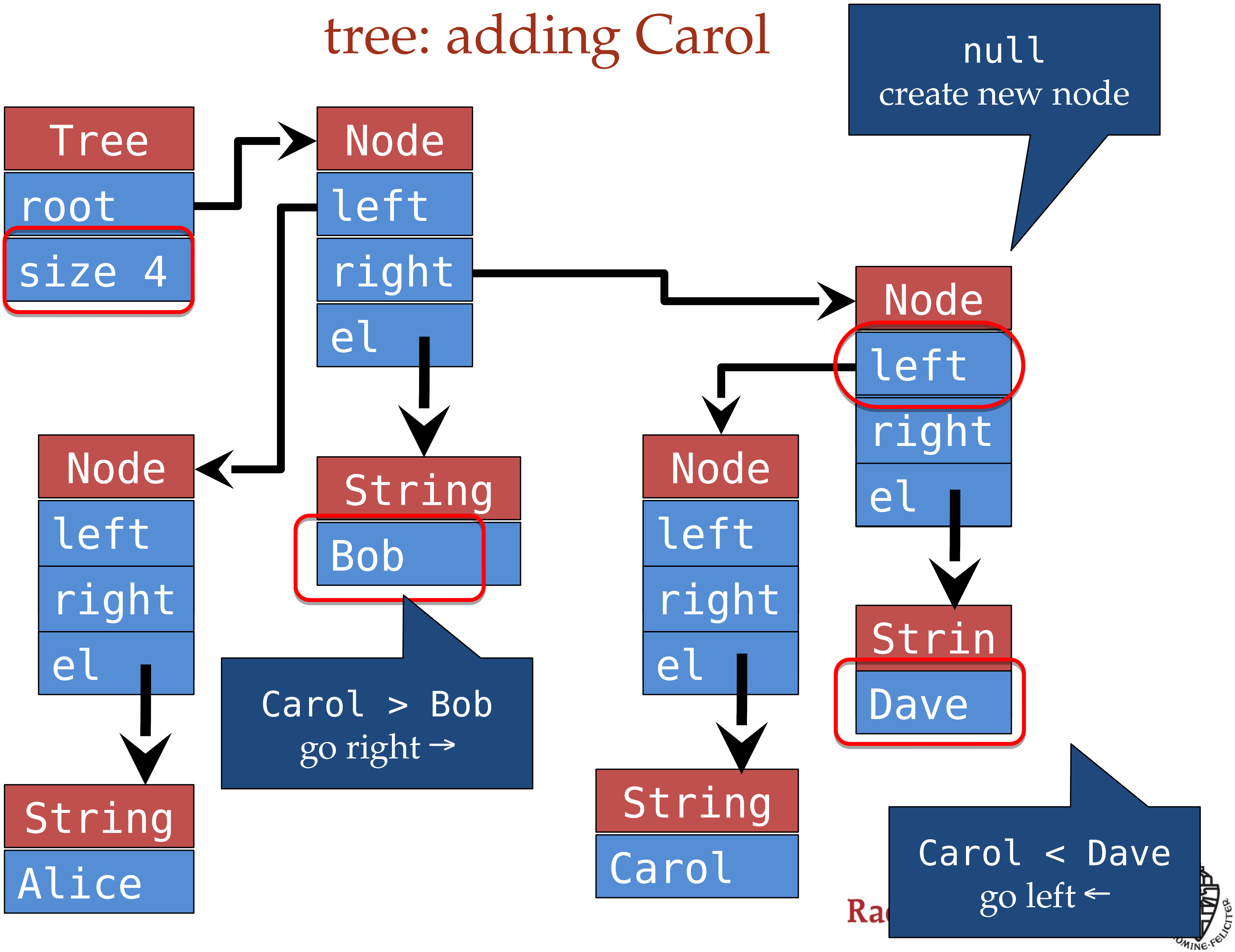
empty subtree: element does not occur

smaller: search left subtree

equal: found

bigger: search right subtree

tree: adding Carol



add to a search tree

```
public boolean add( E e ) {
    if ( root == null ) {
        root = new Node( e );
        return true;
    } else {
        return add( root, e);
    }
}

private boolean add( Node n, E e ) {
    int comp = e.compareTo( n.el );
    if ( comp < 0 ) {
        if ( n.left == null ) {
            n.left = new Node( e );
            return true;
        } else {
            return add( n.left, e );
        }
    } else if ( comp == 0 ) {
        return false;
    } else { // comp > 0
        if (n.right == null) {
            n.right = new Node (e);
            return true;
        } else {
            return add( n.right, e );
        }
    }
}
```

as before:
helper method with
reference to tree

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 (II)

```
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();
    }
```

base class Node

method based counting

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 ? 1 : next.size() + 1 );
    }
}
```

one successor

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;
    }
}
```

two successors

counting the nodes in a tree

- the Tree class has method:

```
public int size() {
    return root == null ? 0 : root.size();
}
```

- the method implementations belong to the subtypes: the dynamic binding mechanism of Java provides that the right version of `size` will be called.

a single recursive method

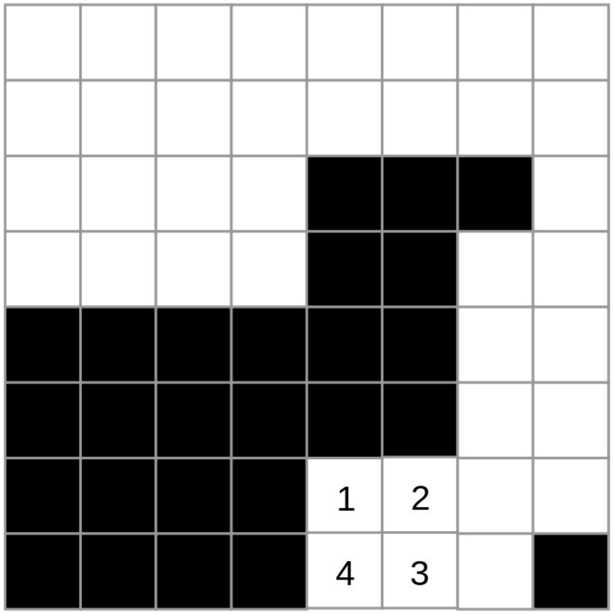
```
public static int size( Node n ) {
    if ( n == null ) {
        return 0;
    } else if ( n instanceof Node0 ) {
        return 1;
    } else if ( n instanceof Node1 ) {
        Node1 n1 = (Node1) n;
        return 1 + size( n1.next );
    } else if ( n instanceof Node2 ) {
        Node2 n2 = (Node2)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

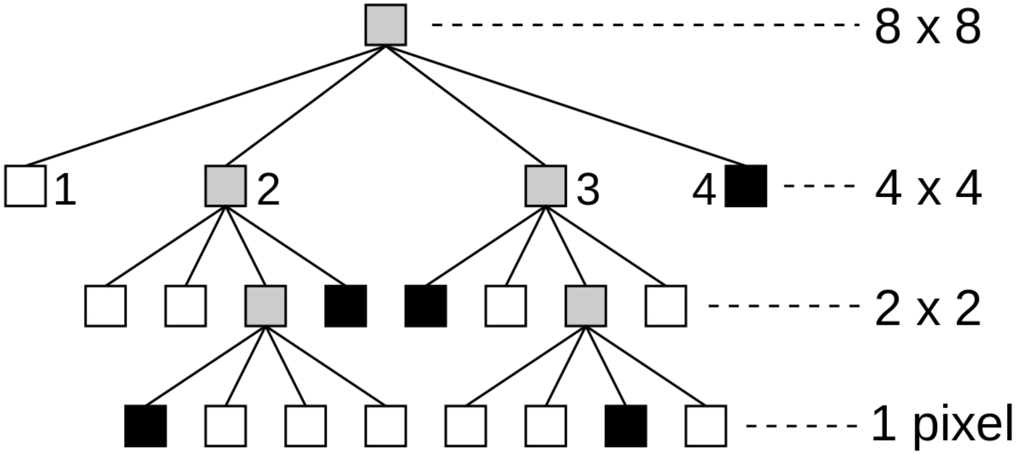
dynamic binding is better

needed type casts are ugly

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 QTNode: 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

0	1
3	2

- operations become recursive methods
  - define operations as methods of the interface
  - make an implementation in each subclass

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)

```java
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

```
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