

Comp 311

Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

September 19, 2019

Recursively Defined Datatypes

Recursively Defined Datatypes

- Case classes allow us to combine multiple pieces of a data into a single object
- But sometimes we don't know how many things we wish to combine
- We can use recursion to define datatypes of unbounded size
- This case corresponds to the Composite Design Pattern

Backus-Naur Form For Lists of Ints

```
List ::= Empty  
      | Cons(Int, List)
```

Examples of Lists

Empty

Cons(3, Empty)

Cons(3, Cons(1, Empty))

Cons(3, Cons(1, Cons(4, Empty)))

Defining Lists With Scala Case Classes

```
sealed abstract class List
case object Empty extends List
case class Cons(head: Int, tail: List) extends List
```

Where Do We Put Functions Over Lists?

- We do not expect to define new subtypes of lists
- We do expect to define many new functions over lists
- Similar to our Case Two Design Template for Abstract Datatypes
- Thus, we will start with our pattern matching template

An Example Function for Lists

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => {  
      if (n == 0) true  
      else containsZero(ys)  
    }  
  }  
}
```


An Example Function for Lists

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => (n == 0) || containsZero(ys)  
  }  
}
```

Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```


Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```

We need to determine our *base case*

Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```



We must determine how to combine these values

Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```

This template is an example of *natural recursion* or *structural recursion*: We recursively decompose and then recombine a computation according to the natural structure of the data.

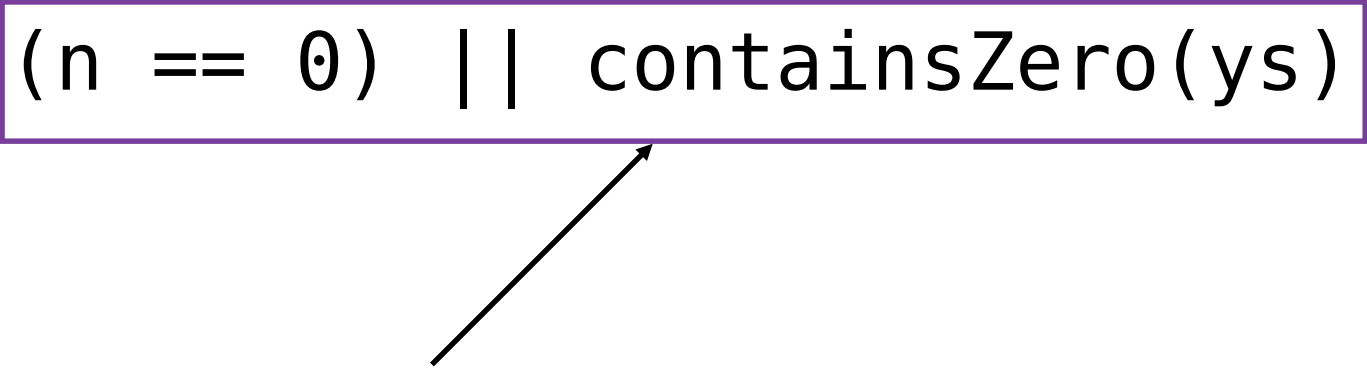
Filling in the Template

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => (n == 0) || containsZero(ys)  
  }  
}
```

Here the base case is easy:
An empty list does not contain zero
(or anything else)

Filling in the Template

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => (n == 0) || containsZero(ys)  
  }  
}
```



We break into cases based on the pieces from match: Either our first element n is zero or the answer lies with the rest of the list

Another Example: How Many Elements?

```
def length(xs: List): Int = {  
  xs match {  
    case Empty => 0  
    case Cons(n, ys) => 1 + length(ys)  
  }  
}
```


Another Example: The Sum of the Elements

```
def sum(xs: List): Int = {  
  xs match {  
    case Empty => 0  
    case Cons(n, ys) => n + sum(ys)  
  }  
}
```

Another Example: The Product of the Elements

```
def product(xs: List): Int = {  
  xs match {  
    case Empty => 1  
    case Cons(n, ys) => n * product(ys)  
  }  
}
```

Converting Hours to Seconds

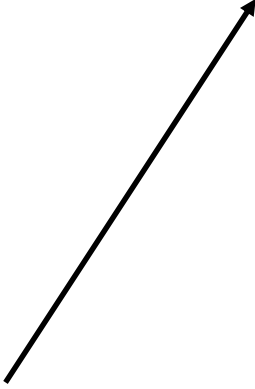
Problem Statement: Given a list of times measured in hours, we want to construct a list of corresponding times measured in seconds

Converting Hours to Seconds

```
def hoursToSeconds(xs: List): List = {  
  xs match {  
    case Empty => Empty  
    case Cons(n, ys) => Cons(seconds(n), hoursToSeconds(ys))  
  }  
}  
  
def seconds(hours: Int) = 3600 * hours
```

Generalizing to a Template

```
def ourFunction(xs: List): List = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => Cons(...n...,  
                             ourFunction(ys))  
  }  
}
```



Really, this is the same template as before, but now Cons is our combining operation

The Natural Numbers

```
Nat ::= 0
      | Next(Nat)
```

The Natural Numbers

```
Nat ::= 0
      | Next(Nat)
```

Here we are between Cases One and Two for Abstract Datatypes:

- No new variants expected
- Many new functions expected
- But some basic functions are intrinsic to the type

Defining The Natural Numbers in Scala

```
sealed abstract class Nat  
case object Zero extends Nat  
case class Next(n: Nat) extends Nat
```


Defining The Natural Numbers in Scala

```
sealed abstract class Nat {  
  def +(n: Nat): Nat  
  def *(n: Nat): Nat  
}
```

Defining The Natural Numbers in Scala

```
case object Zero extends Nat {  
  def +(n: Nat) = n  
  def *(n: Nat) = Zero  
}
```

```
case class Next(n: Nat) extends Nat {  
  def +(m: Nat) = Next(n + m)  
  def *(m: Nat) = m + (n * m)  
}
```

Defining The Natural Numbers in Scala

```
case object Zero extends Nat {  
  def +(n: Nat) = n  
  def *(n: Nat) = Zero  
}
```

Again we have natural
recursion: base case,
recursion, combination

```
case class Next(n: Nat) extends Nat {  
  def +(m: Nat) = Next(n + m)  
  def *(m: Nat) = m + (n * m)  
}
```

Example Reduction

(3 + 2)

Next (Next (Next (Zero)) + Next (Next (Zero))) \mapsto
Next (Next (Next (Zero)) + Next (Next (Zero))) \mapsto
Next (Next (Next (Zero) + Next (Next (Zero)))) \mapsto
Next (Next (Next (Zero + Next (Next (Zero))))) \mapsto
Next (Next (Next (Next (Next (Zero)))))

Factorial

```
def factorial(n: Nat): Nat = {  
  n match {  
    case Zero => Next(Zero)  
    case Next(m) => n * factorial(m)  
  }  
}
```

Transferring The Pattern To Ints

```
def factorial(n: Int): Int = {  
  require (n >= 0)  
  
  if (n == 0) 1  
  else n * factorial(n - 1)  
  
} ensuring (_ > 0)
```

Combining Via Auxiliary Functions

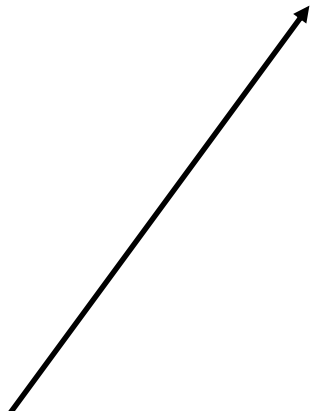
Combining Via Auxiliary Functions

- As our examples with natural numbers shows, it is often desirable to define the *combining operation* of a natural recursion as an auxiliary function
- We can apply this insight to lists and use our template to cover yet more cases

Sorting Lists

```
def sort(xs: List): List = {  
  xs match {  
    case Empty => Empty  
    case Cons(n, ys) => insert(n, sort(ys))  
  }  
}
```

We need to explain how to
insert into a sorted list



Insertion

```
def insert(n: Int, xs: List): List = {  
  xs match {  
    case Empty => Cons(n, Empty)  
    case Cons(m, ys) => {  
      if (n <= m) Cons(n, xs)  
      else Cons(m, insert(n, ys))  
    }  
  }  
}
```

Insertion

```
def insert(n: Int, xs: List): List = {  
  xs match {  
    case Empty => Cons(n, Empty)  
    case Cons(m, ys) => {  
      if (n <= m) Cons(n, xs)  
      else Cons(m, insert(n, ys))  
    }  
  }  
}
```

This parameter is not traversed,
but is used for combination and comparison
Other functions follow this pattern.

Appending Two Lists

```
sealed abstract class List {  
  /**  
   * Returns a new list with the elements of  
   * this list appended to the given list.  
   */  
  def ++(ys: List): List  
}
```

Appending Two Lists

```
case object Empty extends List {  
  def ++(ys: List) = ys  
}
```

Appending Two Lists

```
case class Cons(first: Int, rest: List) extends List {  
  def ++(ys: List) = Cons(first, rest ++ ys)  
}
```

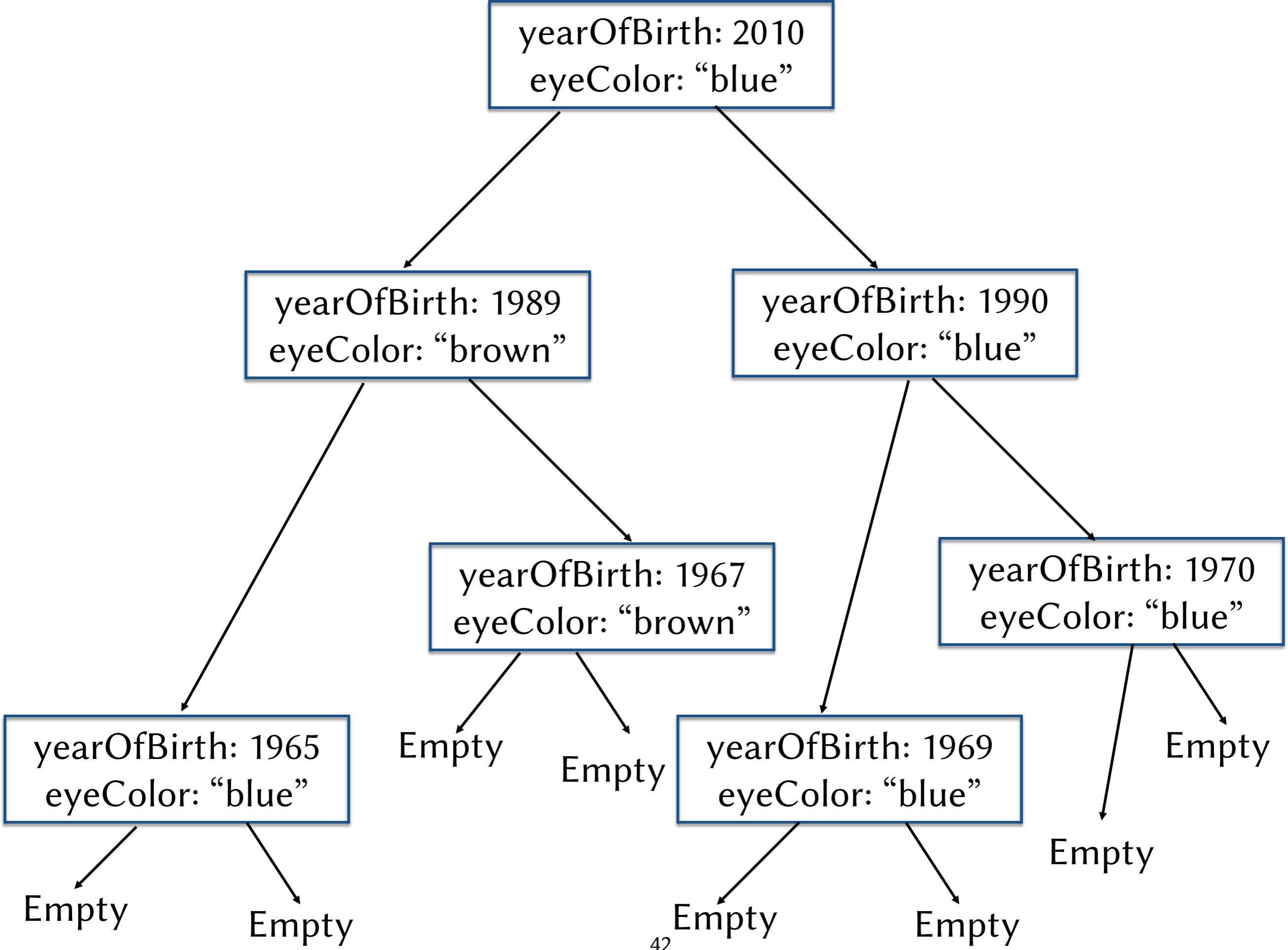
*Recursively Defined
Datatypes with Multiple
Recursive Parameters*

Family Trees

```
TreeNode ::= Empty
           | Child(TreeNode,
                   TreeNode,
                   Int,
                   String)
```


Family Trees

```
sealed abstract class TreeNode  
  
case object EmptyNode extends TreeNode  
  
case class Child(  
  mother: TreeNode,  
  father: TreeNode,  
  yearOfBirth: Int,  
  eyeColor: String)  
  extends TreeNode
```



Family Trees

```
def hasBlueEyedAncestor(t: TreeNode): Boolean =  
  t match {  
    case EmptyNode => false  
    case Child(mother, father, _, eyeColor) =>  
      ( (eyeColor == "Blue")  
        || hasBlueEyedAncestor(mother)  
        || hasBlueEyedAncestor(father) )  
  }
```

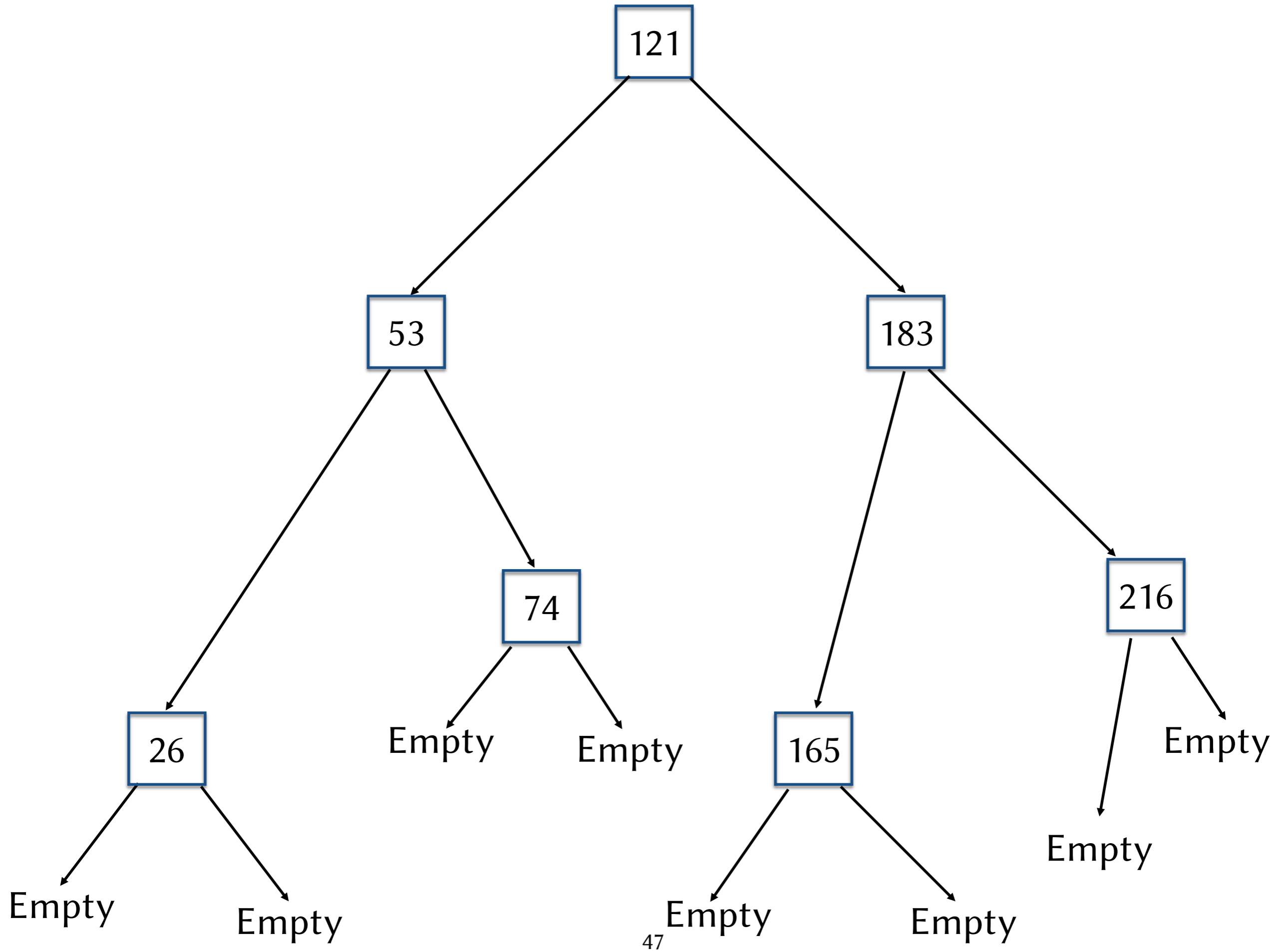
Family Trees

```
def hasBlueEyedAncestor(t: TreeNode): Boolean =  
  t match {  
    case EmptyNode => false  
    case Child(_,_,_,"Blue") => true  
    case Child(mother,father,_,_) =>  
      hasBlueEyedAncestor(mother) ||  
      hasBlueEyedAncestor(father)  
  }
```

Binary Search Trees

Binary Search Trees

- We define trees containing only Ints
- To help us find elements quickly, we abide by the following invariant:
 - At a given node containing value n :
 - All values in the left subtree are less than n
 - All values in the right subtree are greater than n



Binary Search Trees

```
abstract class BinarySearchTree {  
  def contains(n: Int): Boolean  
  def insert(n: Int): BinarySearchTree  
}
```

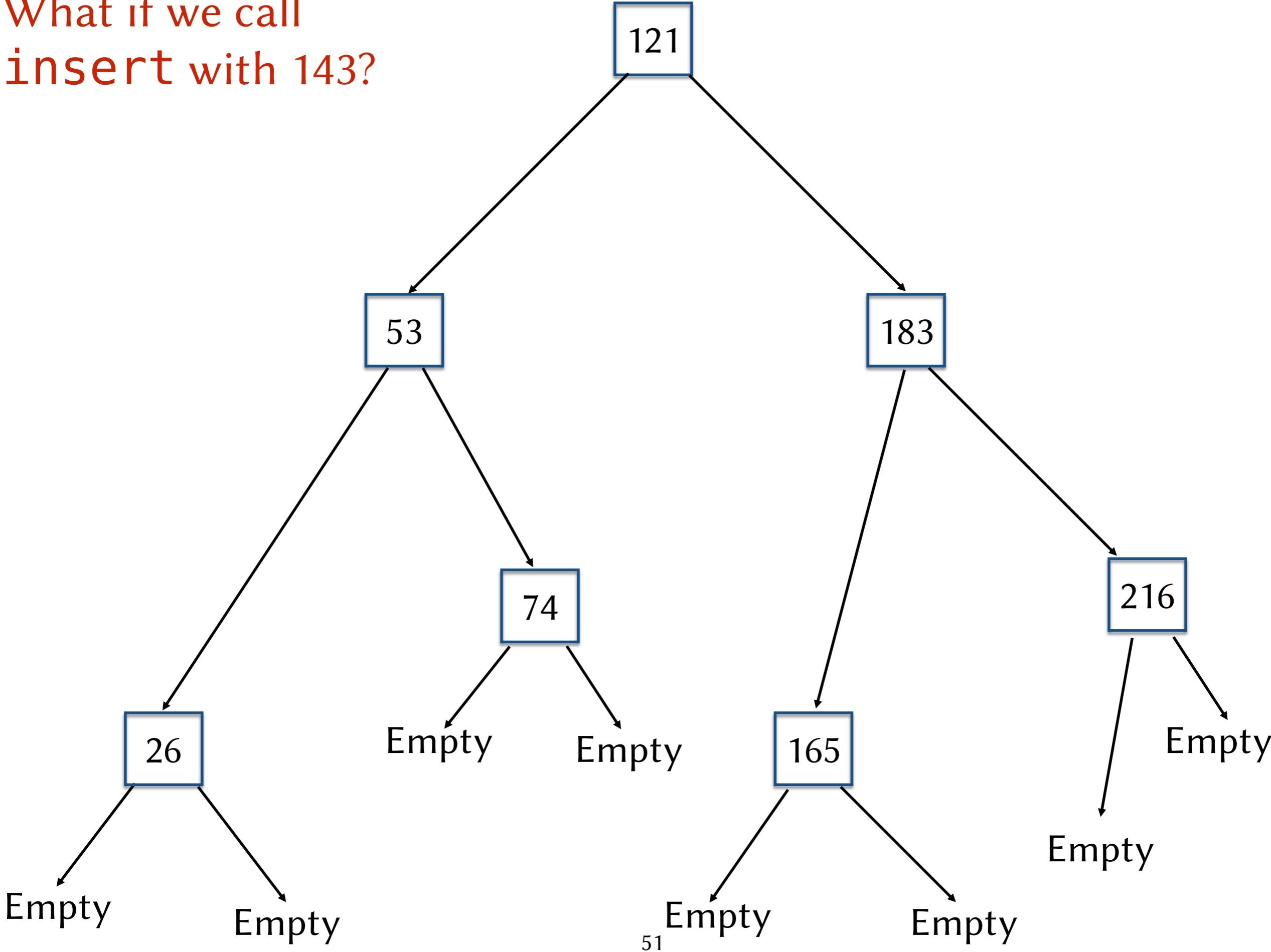

Binary Search Trees

```
case object EmptyTree extends BinarySearchTree {  
  def contains(n: Int) = false  
  def insert(n: Int) = ConsTree(n, EmptyTree, EmptyTree)  
}
```

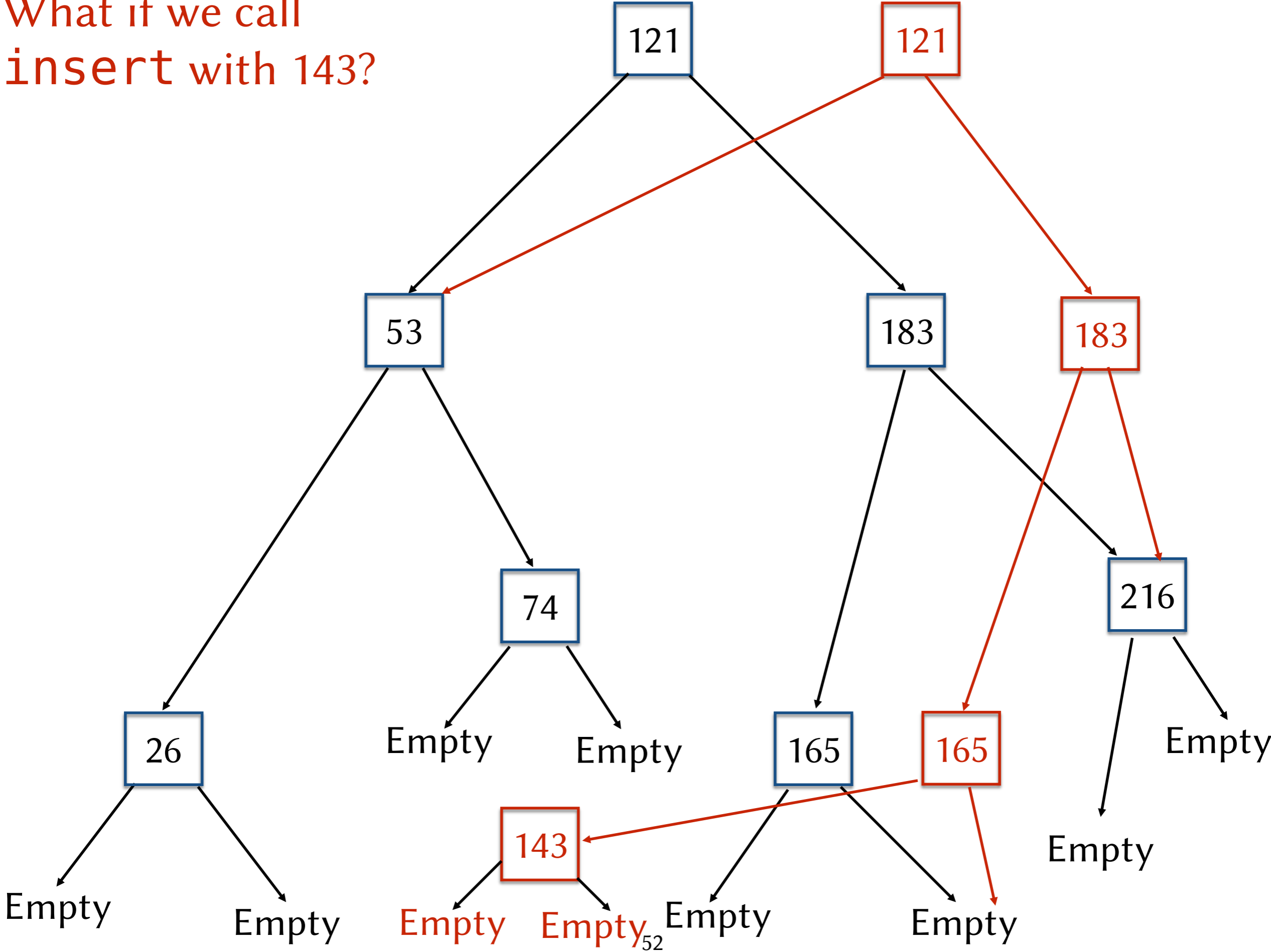
Binary Search Trees

```
case class ConstTree(
  m: Int,
  left: BinarySearchTree,
  right: BinarySearchTree)
extends BinarySearchTree {
def contains(n: Int): Boolean = {
  if (n < m) left.contains(n)
  else if (n > m) right.contains(n)
  else true // n == m
}
def insert(n: Int) = {
  if (n < m) ConstTree(m, left.insert(n), right)
  else if (n > m) ConstTree(m, left, right.insert(n))
  else this // n == m
}
}
```

What if we call
`insert` with 143?



What if we call
insert with 143?



Taking the First Few Elements

```
def take(n: Int, xs: List): List = {  
  require(0 <= n && n <= xs.size)  
  (n, xs) match {  
    case (0, xs) => Empty  
    case (n, Cons(y, ys)) => Cons(y, take(n-1, ys))  
  }  
}
```

Dropping the First Few Elements

```
def drop(n: Int, xs: List): List = {  
  require(0 <= n && n <= xs.size)  
  (n, xs) match {  
    case (0, xs) => xs  
    case (n, Cons(y, ys)) => drop(n-1, ys)  
  }  
}
```

Functional Update of a List

```
def update(xs: List, i: Int, y: Int): List = {  
  require(0 <= i && i < xs.size)  
  assume(xs != Empty) // implied by requirements  
  
  (xs, i) match {  
    case (Cons(z, zs), 0) => Cons(y, zs)  
    case (Cons(z, zs), _) => Cons(z, update(zs, i-1, y))  
  }  
}
```