



ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

# Conc-Trees

Parallel Programming and Data Analysis

Aleksandar Prokopec

## List Data Type

Let's recall the list data type in functional programming.

```
sealed trait List[+T] {  
  def head: T  
  def tail: List[T]  
}  
  
case class ::[T](head: T, tail: List[T])  
  extends List[T]  
  
case object Nil extends List[Nothing] {  
  def head = sys.error("empty list")  
  def tail = sys.error("empty list")  
}
```

## List Data Type

How do we implement a sum method on lists?

## List Data Type

How do we implement a sum method on lists?

```
def sum(lst: List[Int]): Int = lst match {  
  case x :: xs => x + sum(xs)  
  case Nil => 0  
}
```

## List Data Type

How do we implement a `filter` method on lists?

## List Data Type

How do we implement a filter method on lists?

```
def filter[T](lst: List[T])(p: T => Boolean): List[T] = lst match {  
  case x :: xs if p(x) => x :: filter(xs)(p)  
  case x :: xs => filter(xs)(p)  
  case Nil => Nil  
}
```

# Trees

Lists are built for sequential computations – they are traversed from left to right.

# Trees

Lists are built for sequential computations – they are traversed from left to right.

Trees allow parallel computations – their subtrees can be traversed in parallel.

# Trees

Lists are built for sequential computations – they are traversed from left to right.

Trees allow parallel computations – their subtrees can be traversed in parallel.

```
sealed trait Tree[+T]
```

```
case class Node[T](left: Tree[T], right: Tree[T])  
extends Tree[T]
```

```
case class Leaf[T](elem: T) extends Tree[T]
```

```
case object Empty extends Tree[Nothing]
```

## Filter On Trees

How do we implement a filter method on trees?

## Filter On Trees

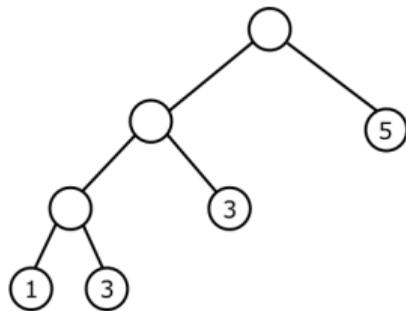
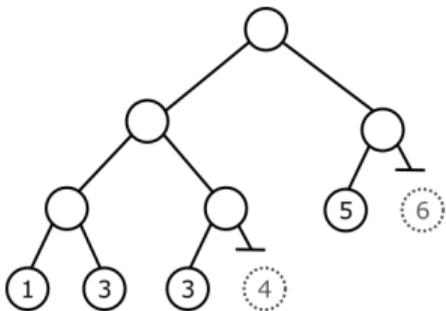
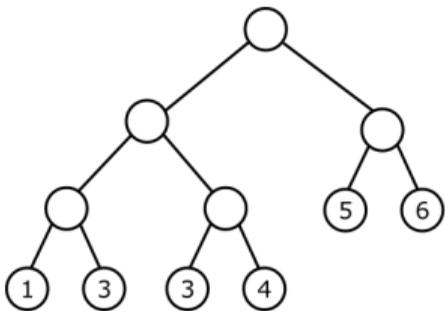
How do we implement a filter method on trees?

```
def filter[T](t: Tree[T])(p: T => Boolean): Tree[T] = t match {  
  case Node(left, right) => Node(parallel(filter(left)(p), filter(right)(p)))  
  case Leaf(elem) => if (p(elem)) t else Empty  
  case Empty => Empty  
}
```

# Filter On Trees

How do we implement a filter method on trees?

```
def filter[T](t: Tree[T])(p: T => Boolean): Tree[T] = t match {  
  case Node(left, right) => Node(parallel(filter(left)(p), filter(right)(p)))  
  case Leaf(elem) => if (p(elem)) t else Empty  
  case Empty => Empty  
}
```



## Conc Data Type

Trees are not good for parallelism unless they are balanced.

## Conc Data Type

Trees are not good for parallelism unless they are balanced.

Let's devise a data type called Conc, which represents balanced trees:

```
sealed trait Conc[+T] {  
  def level: Int  
  def size: Int  
  def left: Conc[T]  
  def right: Conc[T]  
}
```

In parallel programming, this data type is known as the *conc-list* (introduced in the Fortress language).

## Conc Data Type

Concrete implementations of the Conc data type:

```
case object Empty extends Conc[Nothing] {
  def level = 0
  def size = 0
}
class Single[T](val x: T) extends Conc[T] {
  def level = 0
  def size = 1
}
case class <>[T](left: Conc[T], right: Conc[T]) extends Conc[T] {
  val level = 1 + math.max(left.level, right.level)
  val size = left.size + right.size
}
```

## Conc Data Type Invariants

In addition, we will define the following *invariants* for Conc-trees:

1. A  $\langle \rangle$  node can never contain Empty as its subtree.
2. The level difference between the left and the right subtree of a  $\langle \rangle$  node is always 1 or less.

## Conc Data Type Invariants

In addition, we will define the following *invariants* for Conc-trees:

1. A  $\langle \rangle$  node can never contain Empty as its subtree.
2. The level difference between the left and the right subtree of a  $\langle \rangle$  node is always 1 or less.

We will rely on these invariants to implement concatenation:

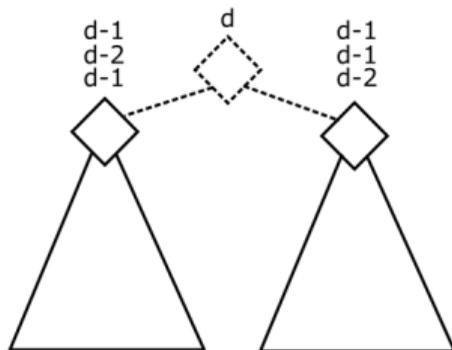
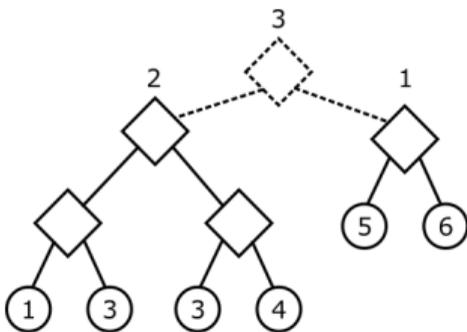
```
def <>(that: Conc[T]): Conc[T] = {  
  if (this == Empty) that  
  else if (that == Empty) this  
  else concat(this, that)  
}
```

## Concatenation with the Conc Data Type

Concatenation needs to consider several cases.

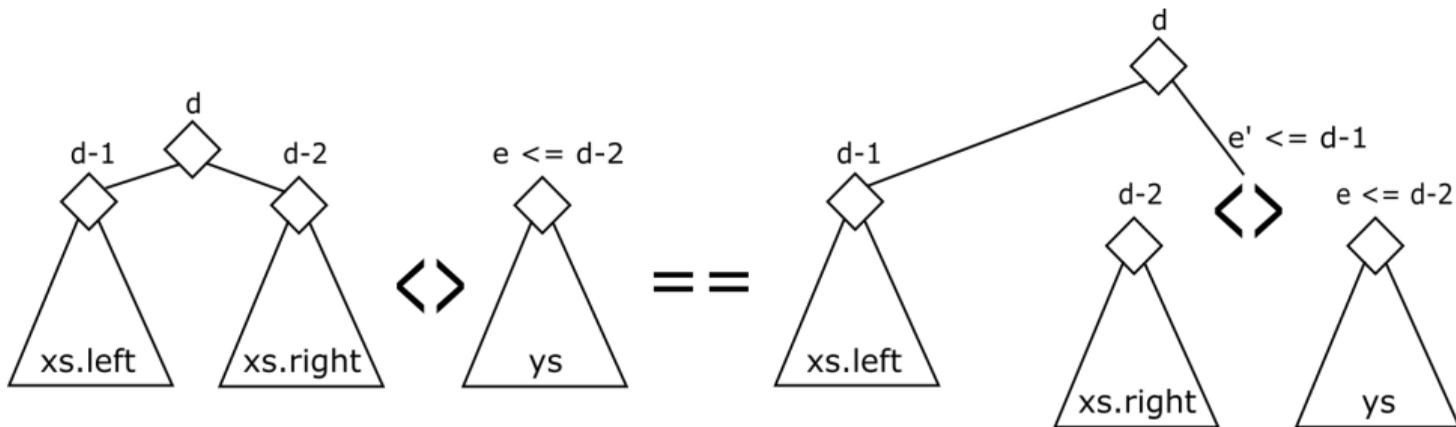
First, the two trees could have height difference 1 or less:

```
def concat[T](xs: Conc[T], ys: Conc[T]): Conc[T] = {  
  val diff = ys.level - xs.level  
  if (diff >= -1 && diff <= 1) new <>(xs, ys)  
  else if (diff < -1) {
```



## Concatenation with the Conc Data Type

Otherwise, let's assume that the left tree is higher than the right one.

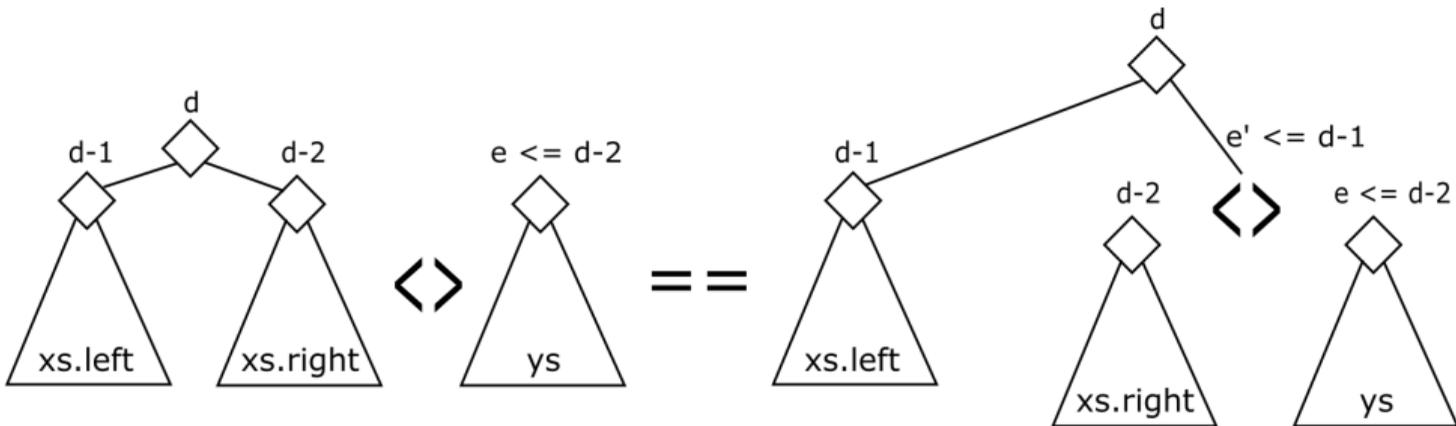


Case 1: The left tree is left-leaning.

Recursively concatenate the right subtree.

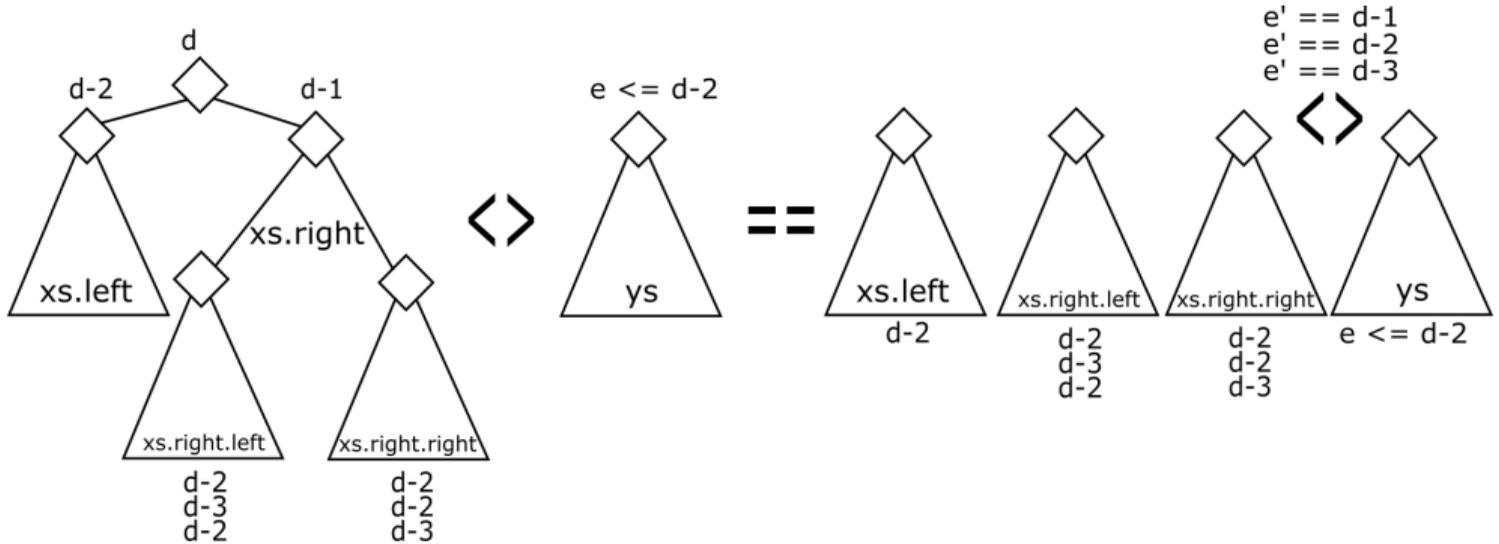
# Concatenation with the Conc Data Type

```
if (xs.left.level >= xs.right.level) {  
  val nr = concat(xs.right, ys)  
  new <>(xs.left, nr)  
} else {
```



# Concatenation with the Conc Data Type

Case 2: The left tree is right-leaning.



## Concatenation with the Conc Data Type

```
} else {  
  val nrr = concat(xs.right.right, ys)  
  if (nrr.level == xs.level - 3) {  
    val nl = xs.left  
    val nr = new <>(xs.right.left, nrr)  
    new <>(nl, nr)  
  } else {  
    val nl = new <>(xs.left, xs.right.left)  
    val nr = nrr  
    new <>(nl, nr)  
  }  
}
```

# Summary

*Question:* What is the complexity of  $\langle \rangle$  method?

- ▶  $O(\log n)$
- ▶  $O(h_1 - h_2)$
- ▶  $O(n)$
- ▶  $O(1)$

## Summary

*Question:* What is the complexity of  $\langle \rangle$  method?

- ▶  $O(\log n)$
- ▶  $O(h_1 - h_2)$
- ▶  $O(n)$
- ▶  $O(1)$

Concatenation takes  $O(h_1 - h_2)$  time, where  $h_1$  and  $h_2$  are the heights of the two trees.