Finger Trees:
a simple general-purpose data structure

RALF HINZE &
ROSS PATERSON

A Case Study by Pierre Gimalac,
Jacob Prud’homme and Chunxiao Wu

Finger Trees:
a simple general-purpose
data structure

JFP **16** (2): 197–217, 2006. © 2005 Cambridge University Press 197
doi:10.1017/S0956796805005769 First published online 16 November 2005 Printed in the United Kingdom

Finger trees:
a simple general-purpose data structure

RALF HINZE
Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

ROSS PATERSON
Department of Computing, City University, London EC1V OHB, UK
(*e-mail:* ross@soi.city.ac.uk)

Abstract

We introduce 2-3 finger trees, a functional representation of persistent sequences supporting access to the ends in amortized constant time, and concatenation and splitting in time logarithmic in the size of the smaller piece. Representations achieving these bounds have appeared previously, but 2-3 finger trees are much simpler, as are the operations on them. Further, by defining the split operation in a general form, we obtain a general purpose data structure that can serve as a sequence, priority queue, search tree, priority search queue and more.

Source

Description and Characteristics

- General-purpose, purely functional base for sequence-like data
- Amortized constant-time access to the ends / appending / removal
- Logarithmic-time concat / split
- Genericity via measure

Structure

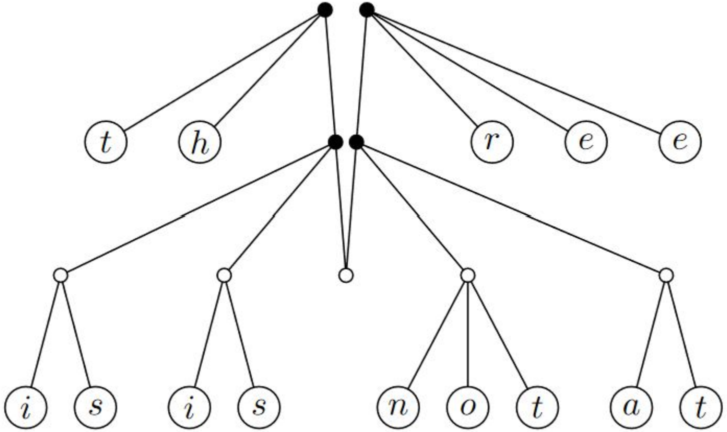
Empty

$\varnothing$

Single

t

Deep



Source

Operations on Finger Trees

- Prepend and Append
- View (left and right)
- Remove (first and last)
- Concatenation
- Split

Prepend and Append

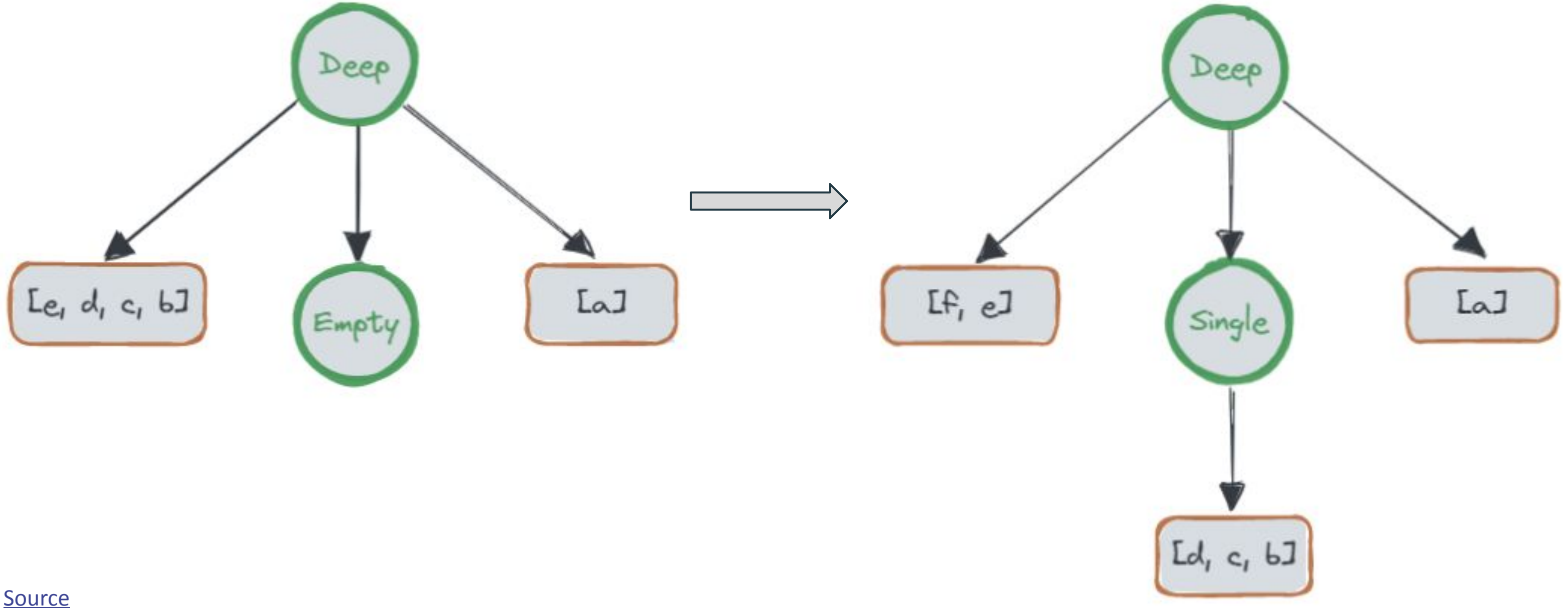
- Symmetric for both ends
- [O(1) amortized time]
- Non-trivial case: when prefix has 4 elements

$$
\begin{array}{ll}
\textbf{infixr } 5 \triangleleft & \\
(\triangleleft) & :: a \rightarrow \mathit{FingerTree}\ a \rightarrow \mathit{FingerTree}\ a \\
a \triangleleft \mathit{Empty} & = \mathit{Single}\ a \\
a \triangleleft \mathit{Single}\ b & = \mathit{Deep}\ [a]\ \mathit{Empty}\ [b] \\
a \triangleleft \mathit{Deep}\ [b, c, d, e]\ m\ \mathit{sf} & = \mathit{Deep}\ [a, b]\ (\mathit{Node3}\ c\ d\ e \triangleleft m)\ \mathit{sf} \\
a \triangleleft \mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf} & = \mathit{Deep}\ ([a] +\!\!+\ \mathit{pr})\ m\ \mathit{sf}
\end{array}
$$

Source

Prepend and Append

- Symmetric for both ends
- [O(1) amortized time]
- Non-trivial case: when prefix has 4 elements



Source

View and Remove (Left and Right)

- Also symmetric for both ends
- O(1) amortized time
- “Remove” is based on “View”

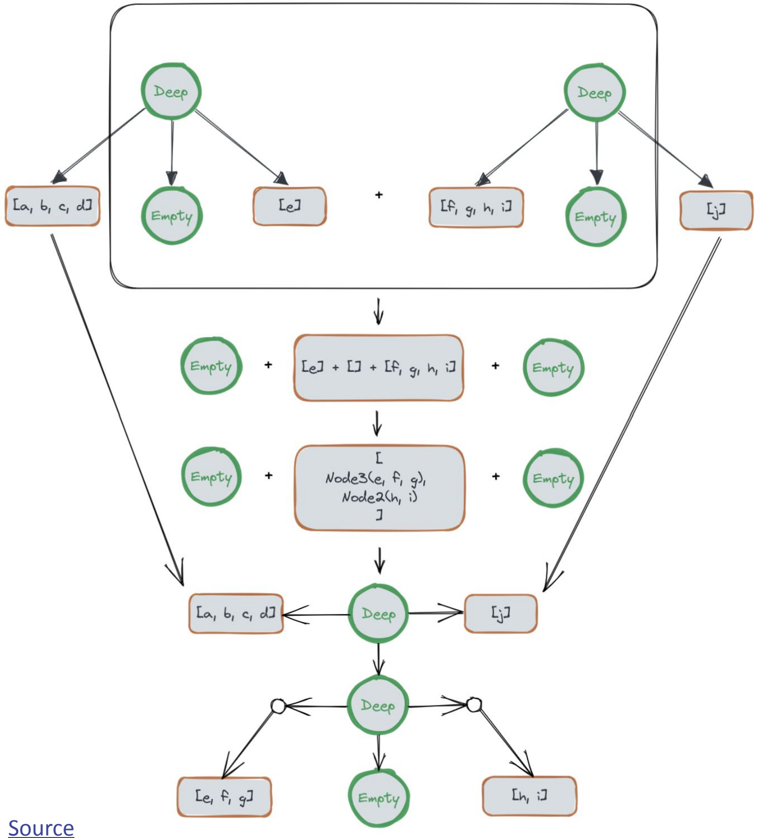
$$
\begin{aligned}
&\mathit{view}_L && :: \mathit{FingerTree}\ a \rightarrow \mathit{View}_L\ \mathit{FingerTree}\ a \\
&\mathit{view}_L\ \mathit{Empty} && = \mathit{Nil}_L \\
&\mathit{view}_L\ (\mathit{Single}\ x) && = \mathit{Cons}_L\ x\ \mathit{Empty} \\
&\mathit{view}_L\ (\mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf}) && = \mathit{Cons}_L\ (\mathit{head}\ \mathit{pr})\ \underline{(\mathit{deep}_L}\ (\mathit{tail}\ \mathit{pr})\ m\ \mathit{sf})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{deep}_L && :: [a] \rightarrow \mathit{FingerTree}\ (\mathit{Node}\ a) \rightarrow \mathit{Digit}\ a \rightarrow \mathit{FingerTree}\ a \\
&\mathit{deep}_L\ []\ m\ \mathit{sf} && = \mathbf{case}\ \mathit{view}_L\ m\ \mathbf{of} \\
& && \qquad \mathit{Nil}_L \qquad\quad \rightarrow \mathit{toTree}\ \mathit{sf} \\
& && \qquad \mathit{Cons}_L\ a\ m' \rightarrow \mathit{Deep}\ (\mathit{toList}\ a)\ m'\ \mathit{sf} \\
&\mathit{deep}_L\ \mathit{pr}\ m\ \mathit{sf} && = \mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf}
\end{aligned}
$$

Source

Concatenation

- Empty case -> trivial
- Single case -> degenerates to appending
- Deep case:
  - Prefix <-> first tree's
  - Suffix <-> second tree's
  - Spine: add both spines, suffix of first tree, and prefix of second tree (shown to right)
- O(logN), N is length of shorter sequence



Source

Measures and Splitting

Measure:

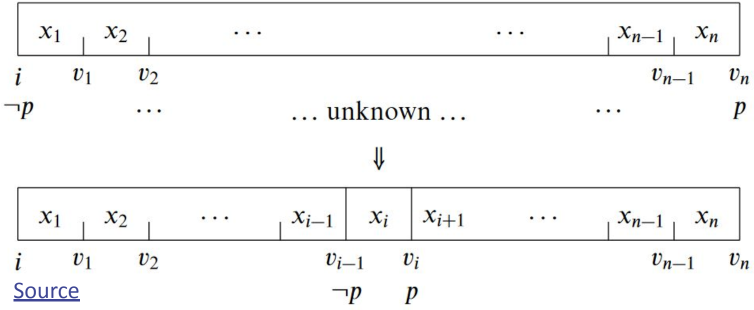
- Attribute on nodes
- Monoid (associative)
- Useful value

Splitting:

- Based on measure
- Happens at point where predicate on measure becomes true
- O(logN), N is length of shorter sequence

Size Measure:

- Gives us random-access
- Identity value: 0
- Size of element: 1
- Predicate: size < idx



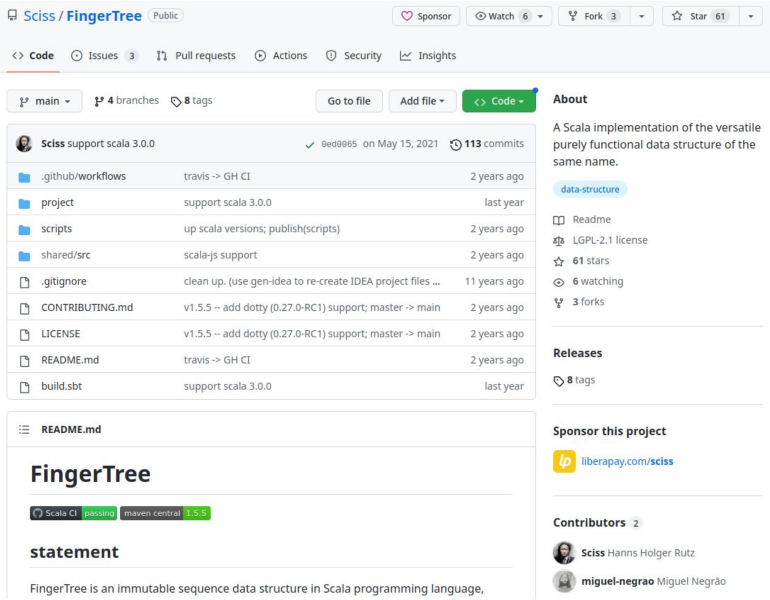
Source

What did we do ?

What did we do ?

- Attempt with an existing library

Finger Trees:
a simple general-purpose
data structure



[Source]

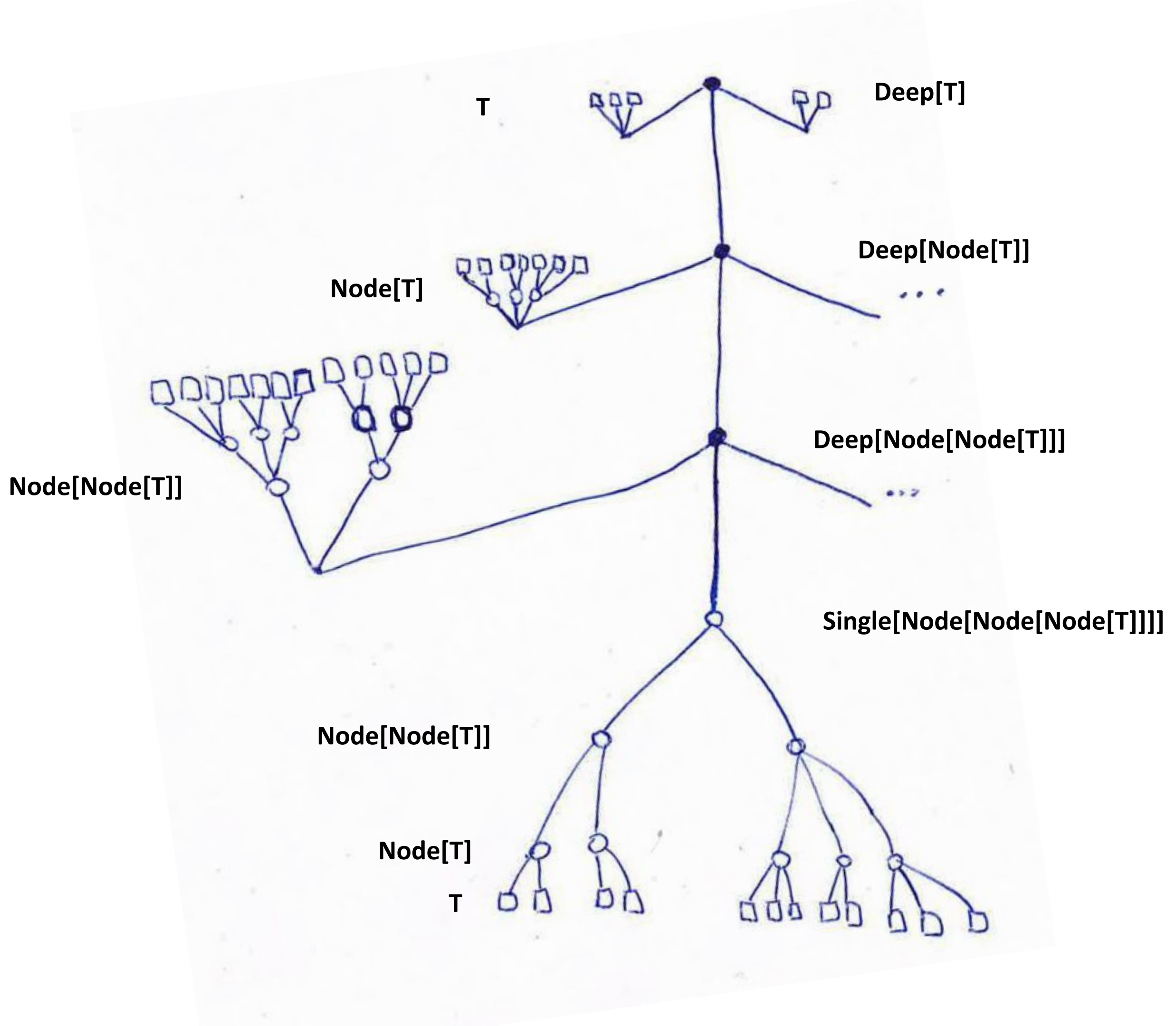
Finger Trees:
a simple general-purpose
data structure

```scala
private sealed trait Node[T]
private final case class Node2[T](a: T, b: T) extends Node[T]
private final case class Node3[T](a: T, b: T, c: T) extends Node[T]

private sealed trait Digit[T]
private final case class Digit1[T](a: T) extends Digit[T]
private final case class Digit2[T](a: T, b: T) extends Digit[T]
private final case class Digit3[T](a: T, b: T, c: T) extends Digit[T]
private final case class Digit4[T](a: T, b: T, c: T, d: T) extends Digit[T]

sealed trait FingerTree[T]
final case class Empty[T]() extends FingerTree[T]
final case class Single[T](value: T) extends FingerTree[T]
final case class Deep[T](
    prefix: Digit[T],
    spine: FingerTree[Node[T]],
    suffix: Digit[T]
) extends FingerTree[T]
```

Source

What did we do ?

- Attempt with an existing library
- Our own (first) implementation


T
Deep[T]
Node[T]
Deep[Node[T]]
Node[Node[T]]
Deep[Node[Node[T]]]
Single[Node[Node[Node[T]]]]
Node[Node[T]]
Node[T]
T

Solution

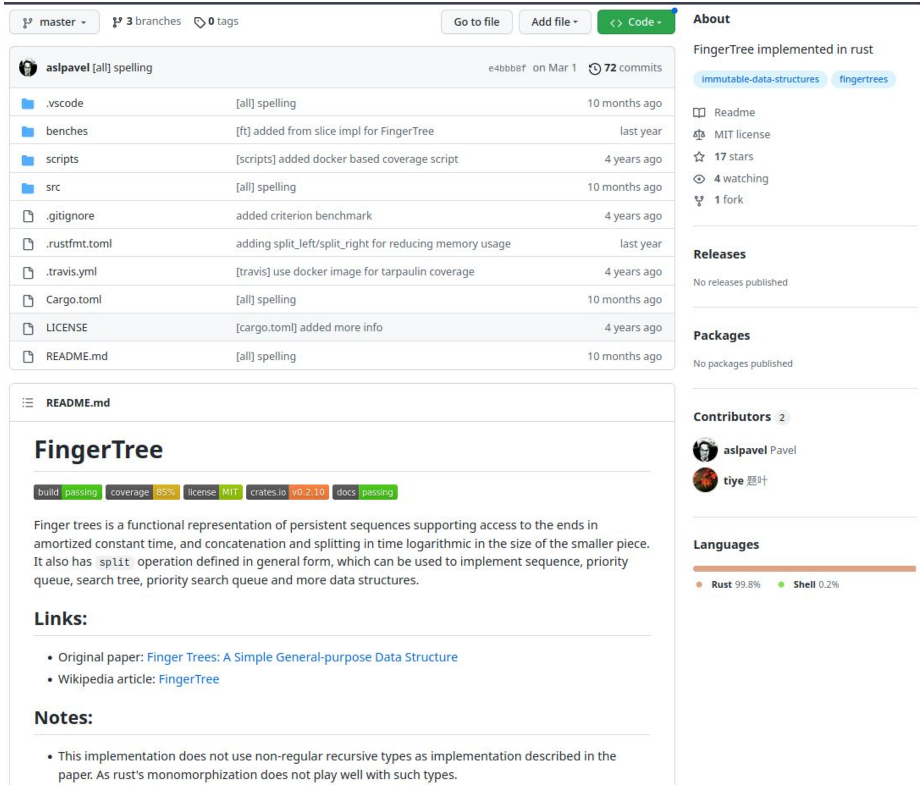
PURELY FUNCTIONAL DATA STRUCTURES

CHRIS OKASAKI
COLUMBIA UNIVERSITY

Source: don't worry about it

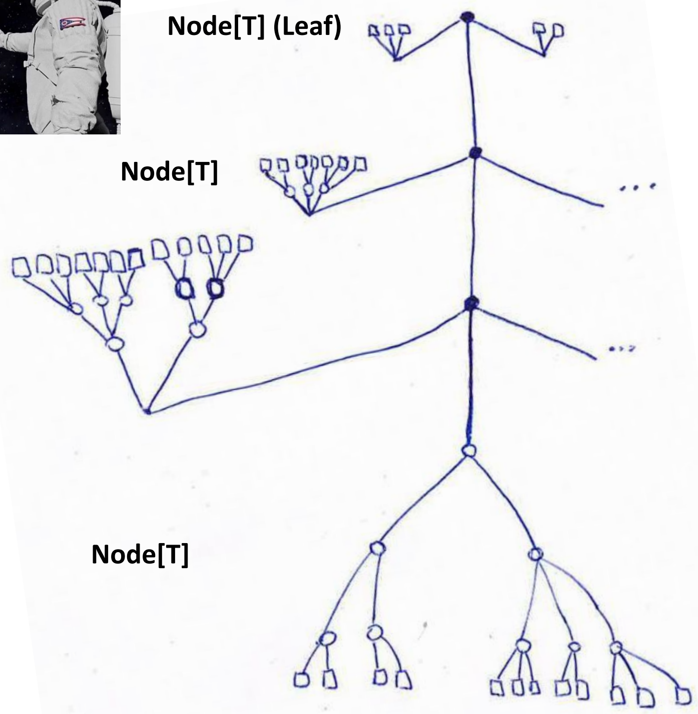
Solution

github.com/aslpavel/fingertree-rs



Finger Trees:
a simple general-purpose
data structure

Finger Trees:
a simple general-purpose
data structure



- 'Leaf(T)' Node
- T -> Node[T]
- Node[Node[...]] -> Node[T]

What did we do ?

- Attempt with an existing library
- Our own (first) implementation
- Rewrite with "simpler" types

Type definition

Finger Trees: a simple general-purpose data structure

Initial version

```scala
sealed trait Node[T]
case class Node2[T](a: T, b: T)
case class Node3[T](a: T, b: T, c: T)

sealed trait Digit[T]
case class Digit1[T](a: T)
case class Digit2[T](a: T, b: T)
case class Digit3[T](a: T, b: T, c: T)
case class Digit4[T](a: T, b: T, c: T, d: T)

sealed trait FingerTree[T]
case class Empty[T]()
case class Single[T](value: T)
case class Deep[T](
  prefix: Digit[T], spine: FingerTree[Node[T]], suffix: Digit[T]
)
```

Source

New version

```scala
sealed trait Node[T]
case class Leaf[T](a: T)
case class Node2[T](left: Node[T], right: Node[T])
case class Node3[T](left: Node[T], middle: Node[T], right: Node[T])

sealed trait Digit[T]
case class Digit1[T](a: Node[T])
case class Digit2[T](a: Node[T], b: Node[T])
case class Digit3[T](a: Node[T], b: Node[T], c: Node[T])
case class Digit4[T](a: Node[T], b: Node[T], c: Node[T], d: Node[T])

sealed trait FingerTree[T]
case class Empty[T]()
case class Single[T](value: Node[T])
case class Deep[T](
  prefix: Digit[T], spine: FingerTree[T], suffix: Digit[T]
)
```

Source

Invariant

Node

```
def isWellFormed(depth: BigInt): Boolean = {
  require(depth >= 0)
  this match
    case Leaf(a) => depth == 0
    case Node2(left, right) =>
      depth != 0
      && left.isWellFormed(depth - 1)
      && right.isWellFormed(depth - 1)
    case Node3(left, middle, right) =>
      depth != 0
      && left.isWellFormed(depth - 1)
      && middle.isWellFormed(depth - 1)
      && right.isWellFormed(depth - 1)
}
```

FingerTree

```
def isWellFormed(depth: BigInt): Boolean = {
  require(depth >= 0)
  this match
    case Empty()       => true
    case Single(value) => value.isWellFormed(depth)
    case Deep(prefix, spine, suffix) =>
      prefix.isWellFormed(depth)
      && suffix.isWellFormed(depth)
      && spine.isWellFormed(depth + 1)
}
```


[Source]()

Examples

Prepend (addL)

```scala
private def addL(value: Node[T], depth: BigInt): FingerTree[T] = {
  require(depth >= 0 && this.isWellFormed(depth) && value.isWellFormed(depth))
  this match {
    case Empty() => Single(value)
    case Single(existingValue) =>
      Deep(Digit1(value), Empty(), Digit1(existingValue))
    case Deep(Digit1(a), spine, suffix) =>
      Deep(Digit2(value, a), spine, suffix)
    case Deep(Digit2(a, b), spine, suffix) =>
      Deep(Digit3(value, a, b), spine, suffix)
    case Deep(Digit3(a, b, c), spine, suffix) =>
      Deep(Digit4(value, a, b, c), spine, suffix)
    case Deep(Digit4(a, b, c, d), spine, suffix) =>
      Deep(Digit2(value, a), spine.addL(Node3(b, c, d), depth + 1), suffix)
  }
}.ensuring(_.isWellFormed(depth))
```

Source

Examples

Concatenation (concat)

```scala
private def toNodes[T](elems: List[Node[T]], depth: BigInt): List[Node[T]] = {
  require(
    depth >= 0
      && elems.size >= 2
      && elems.forall(_.isWellFormed(depth))
  )
  elems match {
    case Nil()                          => ???
    case Cons(a, Nil())                 => ???
    case Cons(a, Cons(b, Nil()))        => List(Node2(a, b))
    case Cons(a, Cons(b, Cons(c, Nil()))) => List(Node3(a, b, c))
    case Cons(a, Cons(b, Cons(c, Cons(d, Nil())))) =>
      List(Node2(a, b), Node2(c, d))
    case Cons(a, Cons(b, Cons(c, tail))) => {
      Cons(Node3(a, b, c), toNodes(tail, depth))
    }
  }
}.ensuring(_.forall(_.isWellFormed(depth + 1)))
```

```scala
private def concat[T](
    tree1: FingerTree[T],
    elems: List[Node[T]],
    tree2: FingerTree[T],
    depth: BigInt
): FingerTree[T] = {
  require(
    depth >= 0
      && tree1.isWellFormed(depth)
      && tree2.isWellFormed(depth)
      && elems.forall(_.isWellFormed(depth))
  )
  decreases(tree1)
  (tree1, tree2) match {
    case (Empty(), _)   => tree2.addL(elems, depth)
    case (Single(e), _) => tree2.addL(elems, depth).addL(e, depth)
    case (_, Empty())   => tree1.addR(elems, depth)
    case (_, Single(e)) => tree1.addR(elems, depth).addR(e, depth)
    case (Deep(prefix1, spine1, suffix1), Deep(prefix2, spine2, suffix2)) =>
      val elemsTree1 = suffix1.toNodeList(depth)
      val elemsTree2 = prefix1.toNodeList(depth)
      forallConcat(elemsTree1, elems, _.isWellFormed(depth))
      forallConcat(elemsTree1 ++ elems, elemsTree2, _.isWellFormed(depth))
      val elemsRec = elemsTree1 ++ elems ++ elemsTree2
      Deep(
        prefix1,
        concat(spine1, toNodes(elemsRec, depth), spine2, depth + 1),
        suffix2
      )
  }
}.ensuring(_.isWellFormed(depth))
```

[Source]

Properties

- isEmpty
- Concatenation (concat)
- Head (headL / headR)
- Prepend (addL)
- Append (addR)
- Tail (tailL / tailR)
- Conversion (toTree / toList)

Properties

- What we have done:

```scala
// EMPTY //
def isEmptyL_law(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.toListL().isEmpty == t.isEmpty
}.holds

def isEmptyR_law(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.toListR().isEmpty == t.isEmpty
}.holds

def emptyConcatL(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.concat(Empty()).toListL() == t.toListL()
}.holds

def emptyConcatR(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  Empty().concat(t).toListR() == t.toListR()
}.holds

def isEmpty_concat(t1: FingerTree[T], t2: FingerTree[T]): Boolean = {
  require(t1.isWellFormed && t2.isWellFormed)
  t1.concat(t2).isEmpty == (t1.isEmpty && t2.isEmpty)
}.holds
```

```scala
// head //
def headL_ListL_head(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.headL == t.toListL().headOption
}.holds

def headL_ListR_last(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.headL == t.toListR().lastOption
}.holds

def headR_ListR_head(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.headR == t.toListR().headOption
}.holds

def headR_ListL_last(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.headR == t.toListL().lastOption
}.holds
```

Properties

- What we have done:

```scala
def concatHeadL(t1: FingerTree[T], t2: FingerTree[T]): Boolean = {
  require(t1.isWellFormed && t2.isWellFormed)
  t1.concat(t2).headL == t1.headL.orElse(t2.headL) because {
    t1 match {
      case Empty() => emptyConcatHeadL(t2)
      case _       => t1.concat(t2).headL == t1.headL
    }
  }
}.holds

def concatHeadR(t1: FingerTree[T], t2: FingerTree[T]): Boolean = {
  require(t1.isWellFormed && t2.isWellFormed)
  t1.concat(t2).headR == t2.headR.orElse(t1.headR) because {
    t2 match {
      case Empty() => emptyConcatHeadR(t1)
      case _       => t1.concat(t2).headR == t2.headR
    }
  }
}.holds

def emptyConcatHeadL(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  Empty().concat(t).headL == t.headL
}.holds

def emptyConcatHeadR(t: FingerTree[T]): Boolean = {
  require(t.isWellFormed)
  t.concat(Empty()).headR == t.headR
}.holds
```

```scala
def addLHeadL(t: FingerTree[T], value: T): Boolean = {
  require(t.isWellFormed)
  t.addL(value).headL.get == value
}.holds

def addRHeadR(t: FingerTree[T], value: T): Boolean = {
  require(t.isWellFormed)
  t.addR(value).headR.get == value
}.holds
```

```scala
// add //
def addL_law(t: FingerTree[T], value: T): Boolean = {
  require(t.isWellFormed)
  t.addL(value).toListL().head == value
}.holds

def addR_law(t: FingerTree[T], value: T): Boolean = {
  require(t.isWellFormed)
  t.addR(value).toListR().head == value
}.holds
```

Finger Trees:
a simple general-purpose
data structure

Properties

- Still in progress:
  Conversion functions: element-preserving and order-preserving

```
// toTree //
def toTree_toListL(l: List[T]): Boolean = {
  check(toTreeL(l).isWellFormed)
  toTreeL(l).toListL() == l
}.holds

def toTree_toListR(l: List[T]): Boolean = {
  toTreeR(l).toListR() == l
}.holds
```

```
// toTree //
def toTreeLR(l: List[T]): Boolean = {
  check(toTreeL(l).isWellFormed)
  check(toTreeR(l).isWellFormed)
  toTreeL(l).content() == toTreeR(l).content()
}.holds

def toTree_toListL(l: List[T]): Boolean = {
  check(toTreeL(l).isWellFormed)
  toTreeL(l).toListL().content == l.content
}.holds

def toTree_toListR(l: List[T]): Boolean = {
  check(toTreeR(l).isWellFormed)
  toTreeR(l).toListR().content == l.content
}.holds
```

Summary

What we have so far

- ✓ Implementation of the finger tree
- ✓ Proofs of properties of most functions

Plan to do next

- ❑ Finish the verification of conversion functions
- ❑ Implement measures and split (?)
- ❑ Prove things about the measure (eg. correctness of split)

Questions?

References

- RALF HINZE and ROSS PATERSON. “Finger trees: a simple general-purpose data structure”. In: Journal of Functional Programming 16.2 (2006), pp. 197–217. doi: `10.1017/S0956796805005769`.

- Chris Okasaki. “Purely functional data structures”. In: 1998.

Appendix

Why are deque operations in amortized constant time?

- Take Prepend (addL) as an example:
    - Non-recursive case is trivial
    - Recursive case:

Operations on Digits of 2 or 3 elements would not propagate to the next level;
while on Digits of 1 or 4 it will propagate, but in doing so it makes the current Digit of size 2 or 3, so that next operation on it would not propagate.

Source