

Finger Trees: a simple general-purpose data structure

RALF HINZE &
ROSS PATERSON

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



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 0HB, 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

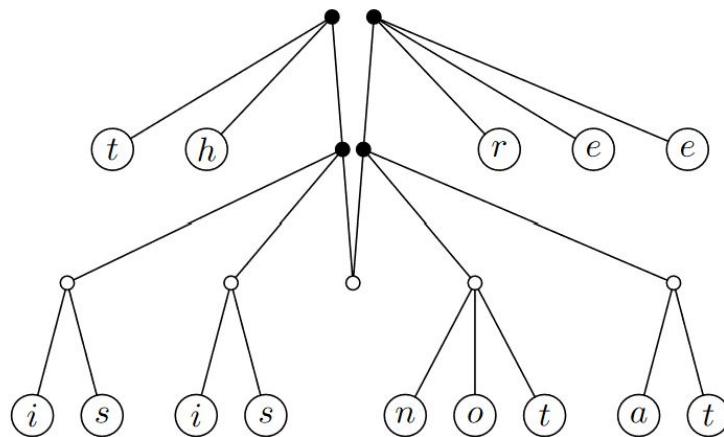
Empty

\emptyset

Single

t

Deep



[Source](#)

Finger Trees:
a simple general-purpose
data structure

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

infixr 5 \triangleleft

$(\triangleleft) \quad :: a \rightarrow FingerTree\ a \rightarrow FingerTree\ a$

$a \triangleleft Empty \quad = Single\ a$

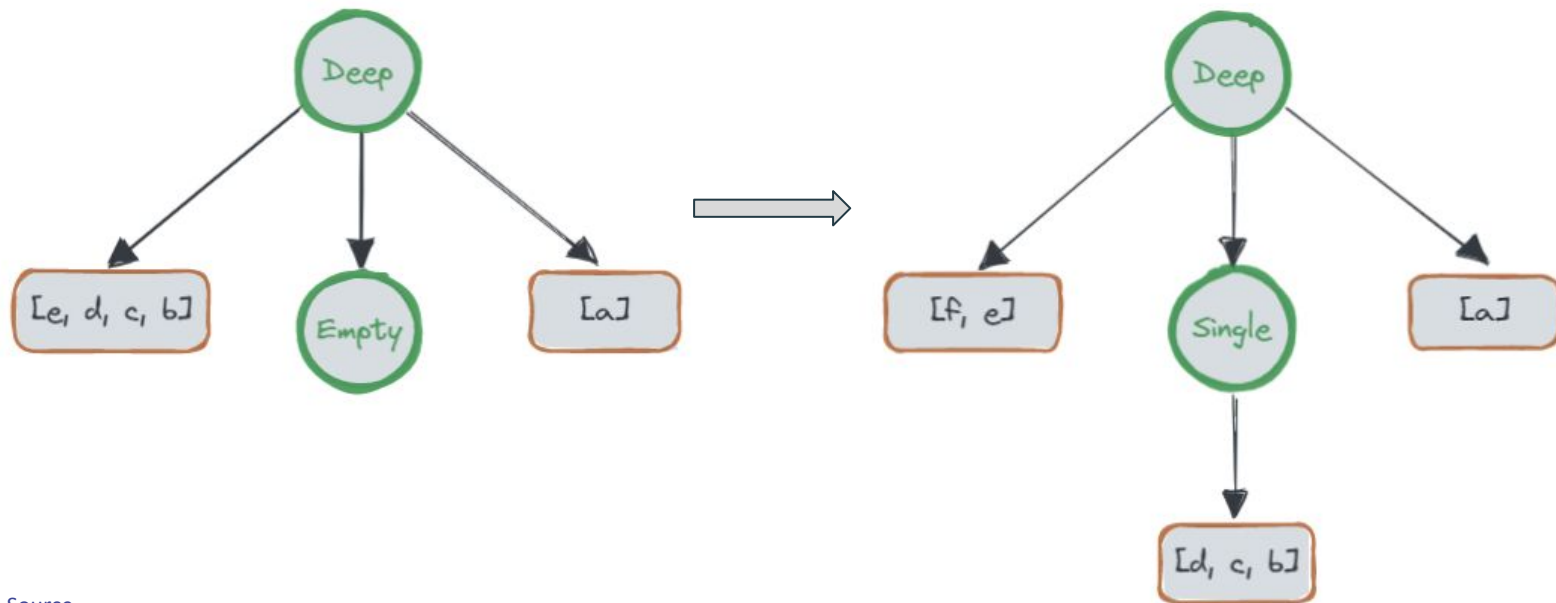
$a \triangleleft Single\ b \quad = Deep\ [a]\ Empty\ [b]$

$a \triangleleft Deep\ [b, c, d, e]\ m\ sf \quad = Deep\ [a, b]\ (Node3\ c\ d\ e\ \triangleleft m)\ sf$

$a \triangleleft Deep\ pr\ m\ sf \quad = Deep\ ([a] \mathrel{++} pr)\ m\ sf$

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”

$view_L :: FingerTree\ a \rightarrow View_L\ FingerTree\ a$

$view_L\ Empty = Nil_L$

$view_L\ (Single\ x) = Cons_L\ x\ Empty$

$view_L\ (Deep\ pr\ m\ sf) = Cons_L\ (head\ pr)\ (deep_L\ (tail\ pr)\ m\ sf)$

$deep_L :: [a] \rightarrow FingerTree\ (Node\ a) \rightarrow Digit\ a \rightarrow FingerTree\ a$

$deep_L\ []\ m\ sf = \mathbf{case}\ view_L\ m\ \mathbf{of}$

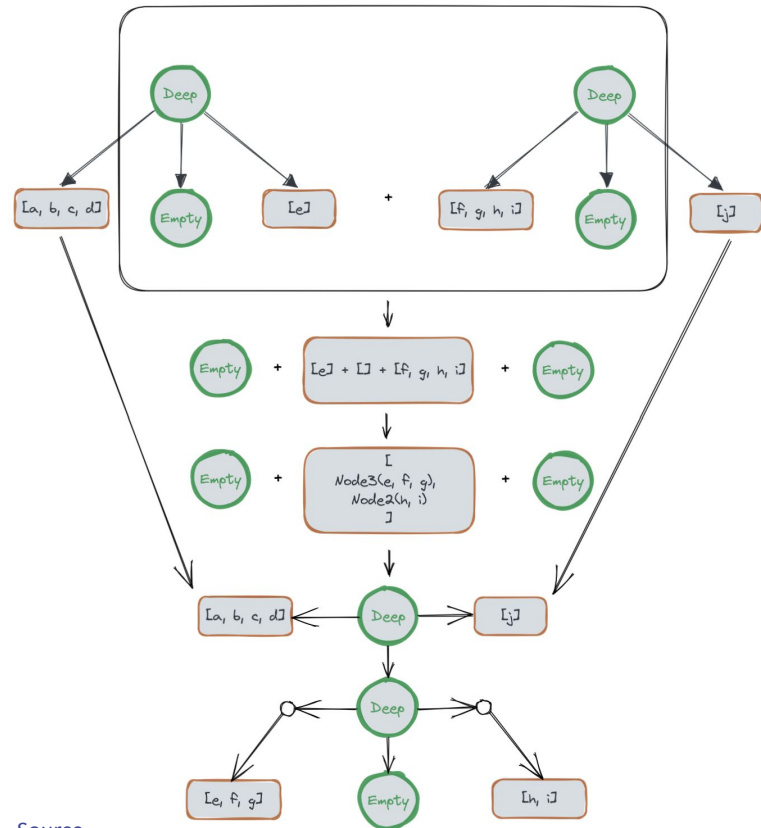
$Nil_L \rightarrow toTree\ sf$

$Cons_L\ a\ m' \rightarrow Deep\ (toList\ a)\ m'\ sf$

$deep_L\ pr\ m\ sf = Deep\ pr\ m\ sf$

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)
- $O(\log N)$, N is length of shorter sequence



[Source](#)

Measures and Splitting

Measure:

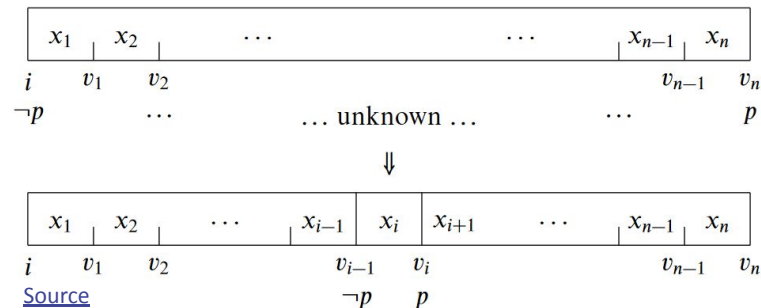
- Attribute on nodes
- Monoid (associative)
- Useful value

Splitting:

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

Size Measure:

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



Finger Trees:
a simple general-purpose
data structure

What did we do ?

Finger Trees: a simple general-purpose data structure

The screenshot shows the GitHub repository for Sciss/FingerTree. The repository is public and has 61 stars, 3 forks, and 6 watchers. It is a Scala implementation of a versatile purely functional data structure. The repository includes a README.md file, a LICENSE file, and a build.sbt file. The README.md file is expanded, showing the title "FingerTree" and a statement: "FingerTree is an immutable sequence data structure in Scala programming language." The repository also has a "data-structure" label and a "Sponsor this project" button.

Sciss / FingerTree Public

<> Code Issues 3 Pull requests Actions Security Insights

main 4 branches 8 tags

Go to file Add file <> Code

Sciss support scala 3.0.0 ✓ 0ed9065 on May 15, 2021 113 commits

| | | |
|-------------------|--|--------------|
| .github/workflows | travis -> GH CI | 2 years ago |
| project | support scala 3.0.0 | last year |
| scripts | up scala versions; publish(scripts) | 2 years ago |
| shared/src | scala-js support | 2 years ago |
| .gitignore | clean up. (use gen-idea to re-create IDEA project files ...) | 11 years ago |
| CONTRIBUTING.md | v1.5.5 -- add dotty (0.27.0-RC1) support; master -> main | 2 years ago |
| LICENSE | v1.5.5 -- add dotty (0.27.0-RC1) support; master -> main | 2 years ago |
| README.md | travis -> GH CI | 2 years ago |
| build.sbt | support scala 3.0.0 | last year |

data-structure

Readme

LGPL-2.1 license

61 stars

6 watching

3 forks

Releases

8 tags

Sponsor this project

liberapay.com/sciss

Contributors 2

Sciss Hanns Holger Rutz

miguel-negrão Miguel Negrão

FingerTree

Scala CI passing Maven Central 1.5.5

statement

FingerTree is an immutable sequence data structure in Scala programming language.

[Source](#)

What did we do ?

- Attempt with an existing library

```
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]

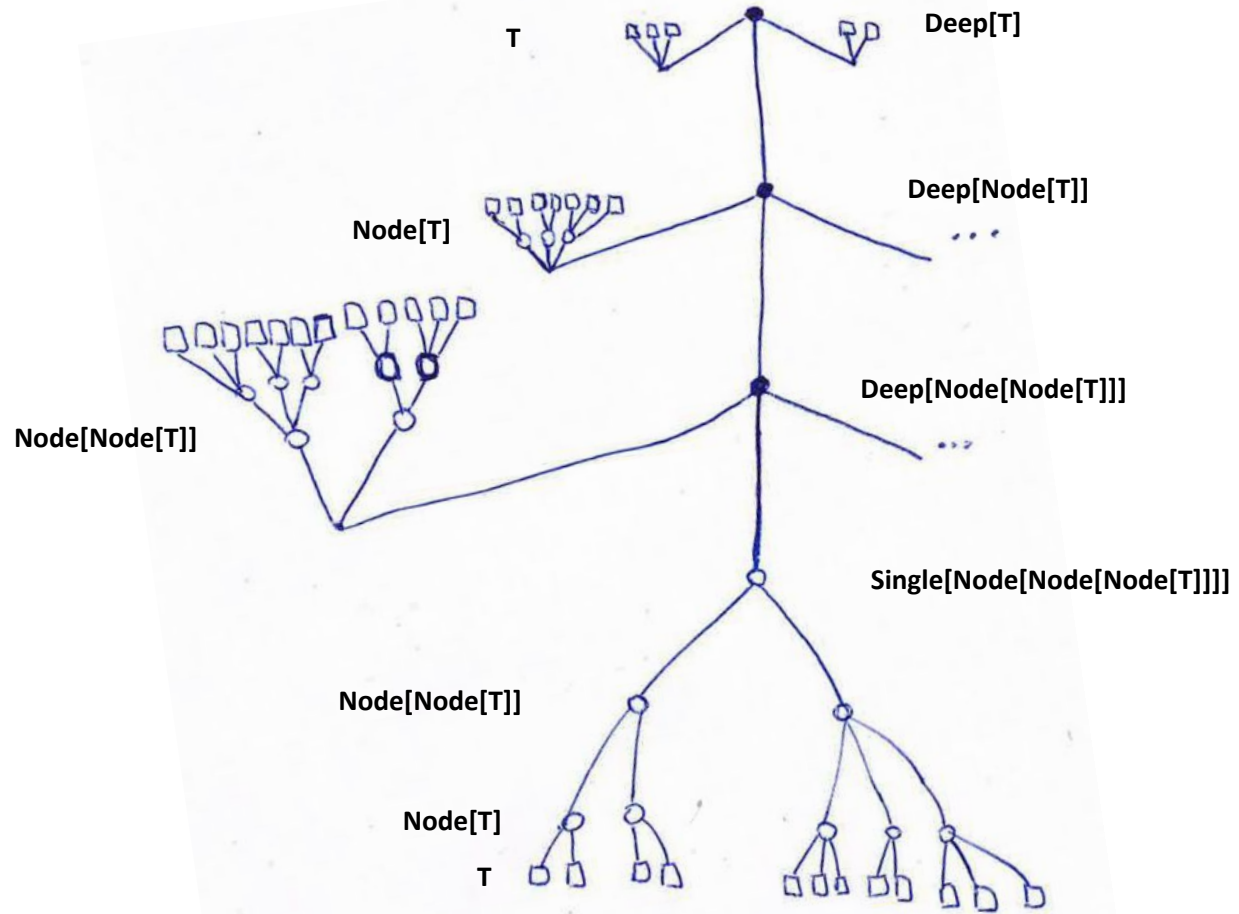
sealed trait FingerTree[T]
final case class Empty[T]() extends FingerTree[T]
final case class Single[T](value: T) extends FingerTree[T]
final case class Deep[T](
  prefix: Digit[T],
  spine: FingerTree[Node[T]],
  suffix: Digit[T]
) extends FingerTree[T]
```

[Source](#)

What did we do ?

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

Finger Trees:
a simple general-purpose
data structure



Solution

PURELY FUNCTIONAL DATA STRUCTURES

CHRIS OKASAKI
COLUMBIA UNIVERSITY

| | |
|--|------------|
| 10 Data-Structural Bootstrapping | 141 |
| 10.1 Structural Decomposition | 142 |
| 10.1.1 Non-Uniform Recursion and Standard ML | 143 |
| 10.1.2 Binary Random-Access Lists Revisited | 144 |
| 10.1.3 Bootstrapped Queues | 146 |

Solution

github.com/aslpavel/fingertree-rs

master 3 branches 0 tags

Go to file Add file <> Code

aslpavel [all] spelling e4bbbef on Mar 1 72 commits

| | | |
|---------------|---|---------------|
| .vscode | [all] spelling | 10 months ago |
| benches | [ft] added from slice impl for FingerTree | last year |
| scripts | [scripts] added docker based coverage script | 4 years ago |
| src | [all] spelling | 10 months ago |
| .gitignore | added criterion benchmark | 4 years ago |
| .rustfmt.toml | adding split_left/split_right for reducing memory usage | last year |
| .travis.yml | [travis] use docker image for tarpaulin coverage | 4 years ago |
| Cargo.toml | [all] spelling | 10 months ago |
| LICENSE | [cargo.toml] added more info | 4 years ago |
| README.md | [all] spelling | 10 months ago |

README.md

FingerTree

build passing
coverage 85%
license MIT
crates.io v0.2.10
docs passing

Finger trees is 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. It also has `split` operation defined in general form, which can be used to implement sequence, priority queue, search tree, priority search queue and more data structures.

Links:

- Original paper: [Finger Trees: A Simple General-purpose Data Structure](#)
- Wikipedia article: [FingerTree](#)

Notes:

- This implementation does not use non-regular recursive types as implementation described in the paper. As rust's monomorphization does not play well with such types.

About

FingerTree implemented in rust

[immutable-data-structures](#)
[fingertrees](#)

Readme
 MIT license
 17 stars
 4 watching
 1 fork

Releases

No releases published

Packages

No packages published

Contributors 2

aslpavel Pavel
 tiye 葉叶

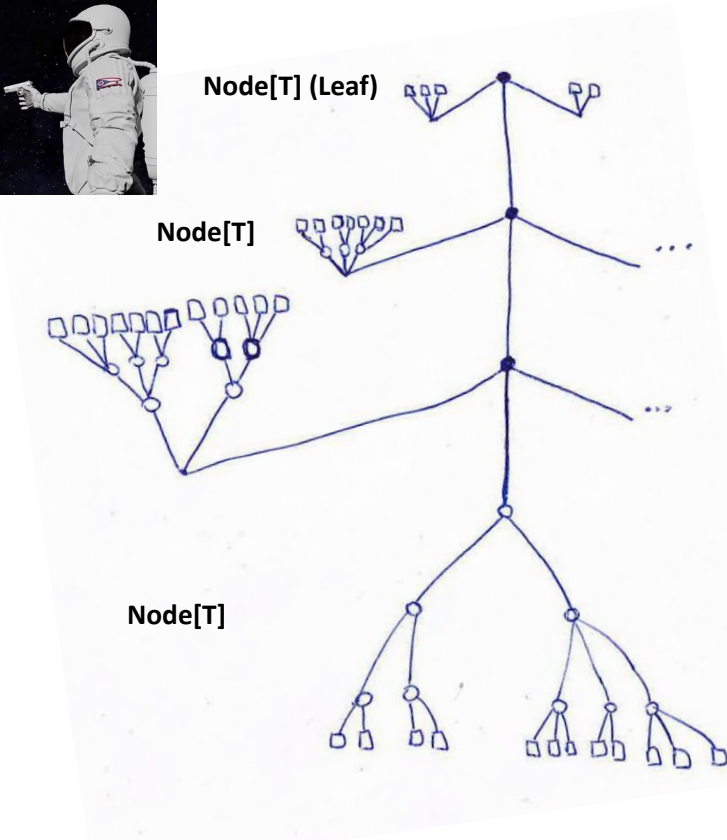
Languages

Rust 99.8%
 Shell 0.2%

Finger Trees:
a simple general-purpose
data structure



Finger Trees:
a simple general-purpose
data structure



- 'Leaf(T)' Node
- $T \rightarrow \text{Node}[T]$
- $\text{Node}[\text{Node}[\dots]] \rightarrow \text{Node}[T]$

What did we do ?

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

Type definition

Initial version

```
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

```
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](#)

Finger Trees:
a simple general-purpose
data structure

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)
}
```

Examples

Prepend (addL)

```
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)

```
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)))
```

```
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](#)

Finger Trees:
a simple general-purpose
data structure

Properties

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

Properties

- What we have done:

```
// 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
```

```
// 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:

```
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
```

```
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
```

```
// 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
```

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.