Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC
Announcements

• Guest Lecture on Tuesday:
  • Shams Imam: Co-routines in Scala

• My office hours next week will be at Thursday 4-5pm
Red-Black Trees Continued
Review: Red-Black Trees

• Every node is colored either red or black

• All leaf nodes are black

• No red node has a red child

• Every path from the root to a leaf contains the same number of black nodes
Review: An Example Red-Black Tree
Review: Strategy for Insertion

- We insert new elements as usual, but then rebalance the tree to maintain the red-black invariants.

- At the end of the rebalancing, we recolor the root to black.

  - This cannot violate our invariants.
Red-Black Trees

abstract class Tree[T <: Ordered[T]] {
  def empty = Leaf[T]
  def contains(x: T): Boolean
  def insert(x: T): Tree[T] = insertChildren(x) match {
    case Branch(c,l,e,r) => Branch(Black, l, e, r)
  }
  def insertChildren(x: T): Branch[T]
}

We call a helper function insertChildren, which performs the insertion and rebalancing.
We take the result from insertChildren, ignore the color of the root and return a tree that is nearly identical except that the root is colored black.
case class Leaf[T <: Ordered[T]]() extends Tree[T] {
  def contains(x: T) = false
  def insertChildren(x: T) = Branch(Red, this, x, this)
}
Red-Black Trees

case class Branch[T <: Ordered[T]]
  (color: Color, left: Tree[T], element: T, right: Tree[T])
extends Tree[T] {

  def contains(x: T) = {
    if (x < element) left contains x
    else if (x > element) right contains x
    else true  // x == element
  }

  ...

}
case class Branch[T <: Ordered[T]]
(color: Color, left: Tree[T], element: T, right: Tree[T])
extends Tree[T] {

...  
def insertChildren(x: T) = {
    if (x < element)
        balance(color, left insertChildren x, element, right)
    else if (x > element)
        balance(color, left, element, right insertChildren x)
    else this
  }
...
}
Rebalancing:
There are Four Cases to Consider
Rebalancing:
There are Four Cases to Consider

We use pattern matching to enumerate the cases.
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
        ... 
    }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
...
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {

    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))

  ...

  }

}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
    ...
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {

    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))

    case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))

    case (Black, Branch(Red, a, x, Branch(Black, c, z, d)), y, c) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))

    case (Black, Branch(Black, a, x, Branch(Black, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))

    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {

  case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
    Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))

  case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
    Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))

...
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case _ =>
      ...
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
      ...
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
        case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
            Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
        case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
            Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
        case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
            Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
        case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
            Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
        ...
    }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    ...
  }
}
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
  (c, l, x, r) match {
    case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
      Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
    case _ => Branch(c, l, x, r)
  }
}
Red-Black Trees

case class Branch[T <: Ordered[T]]
  (color: Color, left: Tree[T], element: T, right: Tree[T])
extends Tree[T] {
  ...
  def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case _ => Branch(c, l, x, r)
    }
  }
  ...
}
case class Branch[T <: Ordered[T]]
  (color: Color, left: Tree[T], element: T, right: Tree[T]) extends Tree[T] {

  def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case _ => Branch(c, l, x, r)
    }
  }

  ...

  ...

  In some languages (such as ML) we could factor this out with “or” patterns
Discussion

• This implementation of red-black trees is dramatically simpler than most imperative approaches:
  
  • Imperative approaches typically include eight cases, branching on the color of the red parent’s sibling

  • These cases help to avoid some assignment and copying in an imperative setting
Streams
Streams

- Streams are a form of “lazy” sequence

- Inspired by signal-processing systems (such as digital circuits):
  - Components accept *streams* of signals as input, transform their input, and produce streams of signals as outputs
abstract class Stream[+T] {
  def head(): T
  def tail(): Stream[T]
  def map[S](f: T => S): Stream[S]
  def flatMap[S](f: T => Stream[S]): Stream[S]
  def ++[S >: T](that: Stream[S]): Stream[S]
  def withFilter(f: T => Boolean): Stream[T]
  def nth(n: Int): T
}
Streams

case object NilStream extends Stream[Nothing] {
  def head() = throw new Error()
  def tail() = throw new Error()
  def map[S](f: Nothing => S): Stream[S] = NilStream
  def flatMap[S](f: Nothing => Stream[S]): Stream[S] = NilStream
  def ++[S >: Nothing](that: Stream[S]) = that
  def withFilter(f: Nothing => Boolean) = NilStream
  def nth(n: Int) = throw new Error()
}
case class ConsStream[+T](head: T, _tail: () => Stream[T]) extends Stream[T] {
    def tail = _tail()
    def map[S](f: T => S): Stream[S] =
        ConsStream(f(head), () => (tail map f))
    def flatMap[S](f: T => Stream[S]): Stream[S] =
        f(current) ++ tail.flatMap(f)
    def ++[S >: T](that: Stream[S]): Stream[S] =
        ConsStream(head, () => tail ++ that)
    ...
}
case class ConsStream[+T](head: T, _tail: () => Stream[T]) extends Stream[T] {
  ...
  def withFilter(f: T => Boolean) = {
    if (f(head)) ConsStream(head, () => tail.withFilter(f))
    else tail.withFilter(f)
  }
  def nth(n: Int) = {
    require (n >= 0)
    if (n == 0) head
    else tail.nth(n - 1)
  }
}
def range(low: Int, high: Int): Stream[Int] = 
  if (low > high) NilStream 
  else ConsStream(low, () => range(low + 1, high))
def intsFrom(n: Int): Stream[Int] = ConsStream(n, () => intsFrom(n + 1))
Streams

```scala
val nats = intsFrom(0)
```
def fibGen(a: Int, b: Int): Stream[Int] = ConsStream(a, () => fibGen(b, a + b))
Streams

val fibs = fibGen(0, 1)
def push(x: Int, ys: Stream[Int]) = {
    ConsStream(x, () => ys)
}
def isDivisible(m: Int, n: Int) = (m % n == 0)
Streams

def isDivisible(m: Int, n: Int) = (m % n == 0)

val noSevens = nats withFilter (isDivisible(_, 7))
def sieve(stream: Stream[Int]): Stream[Int] =
  ConsStream(stream.head,
             () => sieve(stream.tail withFilter
                           (x => !(isDivisible
                                  (x, stream.head))))))
A Stream of Primes

val primes = sieve(intsFrom(2))
A Stream of Primes

> primes.head
res5: Int = 2

> primes.nth(1)
res6: Int = 3

> primes.nth(2)
res7: Int = 5

> primes.nth(3)
res8: Int = 7
def add(xs: Stream[Int], ys: Stream[Int]): Stream[Int] = {
  (xs, ys) match {
    case (NilStream, _) => ys
    case (_, NilStream) => xs
    case (ConsStream(x, f), ConsStream(y, g)) =>
      ConsStream(x + y, () => add(f(), g()))
  }
}
Streams

def ones(): Stream[Int] = ConsStream(1, ones)
Alternative Definition of the Stream of Natural Numbers

```python
def nats(): Stream[Int] =
    ConsStream(0, () => add(ones, nats))
```
Alternative Definition of the Fibonacci Stream

def fibs(): Stream[Int] =
    ConsStream(0,
        () => ConsStream(1,
            () => add(fibs.tail, fibs)))
def scaleStream(c: Int, stream: Stream[Int]): Stream[Int] = 
  stream map (_ * c)

def powersOfTwo(): Stream[Int] = 
  ConsStream(1, () => scaleStream(2, powersOfTwo))
Alternative Definition of the Stream of Primes

def primes() =
    ConsStream(2, () => intsFrom(3) withFilter isPrime)

def isPrime(n: Int): Boolean = {
    def iter(next: Stream[Int]): Boolean = {
        if (square(next.head) > n) true
        else if (isDivisible(n, next.head)) false
        else iter(next.tail)
    }
    iter(primes)
}
Numeric Integration with Streams

\[ S_i = c + \sum_{j=1}^{i} x_j \, dt \]
def integral(integrand: Stream[Double], init: Double, dt: Double) = {
  def inner(): Stream[Double] = {
    ConsStream(init,
      () => addStreams(scaleStream(dt, integrand),
                       inner))
  }
  inner
}
Streams and Local State

def withdraw(balance: Int, amounts: Stream[Int]): Stream[Int] = {
    ConsStream(balance,
        () => withdraw(balance - amounts.head,
                        amounts.tail))
}
Discussion

• Our modeling of a bank account is a purely functional program without state

• Nevertheless:
  • If a user provides the stream of withdrawals, and
  • The stream of balances is displayed as outputs,
  • The system will behave from a user’s perspective as a stateful system
Discussion

• The key to understanding this paradox is that the “state” is in the world:
  • The user/bank system is stateful and provides the input stream
  • If we could “step outside” our own perspective in time, we could view our withdrawal stream as another stateless stream of transactions