Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC
Homework 3 Workload
Homework 3 Course Pace

Frequency

Rating

0  5  10  15
1  2  3  4  5
Actions

• Next year: Switch the order of Assignments 2 and 3
• Make the link to abstract datatypes more explicit
• Minimize constraints on file and directory layouts
Comments

- Design vs functional programming
- Correctness points on homeworks
Functional Data Structures
Leftist Heaps
Leftist Heaps

- Often in a collection of elements we only need to access the *minimum* element.

- A data structure that supports access only to the minimum element is called a *heap*:
  - A tree in which the element at the root of each subtree is the minimum element of that subtree.
Leftist Heaps

• Let the rank of a node be the length of its right spine

• Then a leftist heap also satisfies the following property:
  • The rank of a left child is no smaller than the rank of its sibling
Consequences of the Leftist Property

• The right spine of a node is always the shortest path to a leaf

• The right spine of a node contains $O(\log n)$ elements in the worst case

• The elements along the right spines are in sorted order
Efficient Merging of Two Leftist Heaps

- Intuitively, we can merge two leftist heaps by:
  - Merging their right spines as if they were sorted lists
  - Swapping child nodes along the merged right spine as needed to preserve the leftist property
Leftist Heaps

abstract class Heap[T <: Ordered[T]] {
  def empty = Leaf[T]
  def isEmpty: Boolean

  def insert(element: T): Heap[T] = this merge Branch(1, element, empty, empty)
  def merge(that: Heap[T]): Heap[T]
  /* require (! isEmpty) */
  def min: T
  /* require (! isEmpty) */
  def deleteMin: Heap[T]

  def rank: Int

  def makeBranch(x: T, a: Heap[T], b: Heap[T]) = {
    if (a.rank >= b.rank) Branch(b.rank + 1, x, a, b)
    else Branch(a.rank + 1, x, b, a)
  }
}
Leftist Heaps

case class Leaf[T <: Ordered[T]]() extends Heap[T] {
  def rank = 0
  def isEmpty = true

  def merge(that: Heap[T]) = that

  def min = throw new Error("Attempt to call min on an empty heap")
  def deleteMin = throw new Error("Attempt to call deleteMin on an empty heap")
}
Leftist Heaps

case class Branch[T <: Ordered[T]](rank: Int, x: T, left: Heap[T], right: Heap[T])
extends Heap[T] {
  def isEmpty = false

  def merge(that: Heap[T]) = {
    that match {
      case Leaf() => this
      case Branch(_, y, l, r) =>
        if (x <= y) makeBranch(x, left, right merge that)
        else makeBranch(y, l, this merge r)
    }
  }

  def min = x
  def deleteMin = left merge right
}
Red-Black Trees
Red-Black Trees

• With naive binary search trees, lookup can take $O(n)$ time in the worst case

• We can fix this problem by rebalancing the trees as we add elements

• Red-Black trees are one approach to keeping the trees approximately balanced
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
An Example Red-Black Tree
Red-Black Trees

• These invariants imply that:

  • The longest path from the root to a leaf consists of an alternating sequence of red nodes and black nodes

  • The shortest path from the root to a leaf consists of all black nodes

• Thus, there is at most a factor of two difference in length between the shortest and longest paths
Red-Black Trees

sealed abstract class Color

case object Red extends Color

case object Black extends Color
Red-Black Trees

sealed abstract class Color

  case object Red extends Color
  case object Black extends Color

All subclasses of a sealed class must be defined in the same file as the sealed class.
Red-Black Trees

sealed abstract class Color
  case object Red extends Color
  case object Black extends Color

Pattern matching against a sealed class is checked to ensure exhaustiveness.
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.
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 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)
}
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

• Because the base case of insertChildren (at a leaf node) always inserts a red node, the number of black nodes along each path is unaffected.

• However, the new tree might contain a red node with a red child.