objc.io | objc 中国

集合类型优化

已对应 Swift 4

Károly Lőrentey 著
王巍, 陈聿菡 译

前言

1 引言

2 有序数组 (Sorted Arrays)

3 将 NSOrderedSet Swift 化

4 红黑树

5 写时复制 (Copy-On-Write) 优化

6 B 树

7 额外优化

8 总结

这本书是如何创建的

前言

从表面上来说，这本书围绕如何实现高性能集合类型进行展开。针对同一个简单的问题，我们将提供多种不同的解决方案，并依次对它们进行详细地说明。同时为了不断地挑战性能巅峰，我们会一直走在找寻新方法的探索征途中。

Swift 提供了很多工具来帮助我们表达自己的想法，说句心里话，这本书仅仅只是针对这些工具进行了一个愉快的探索。本书不会告诉你如何创造一个卓越的 iPhone 应用；但是它会教给你一些工具和技术，以帮助你更好地借助 Swift 代码的形式来表达想法。

这本书源于我曾为 dotSwift 2017 大会 准备的大量笔记和代码。由于准备的材料过于详实有趣，我无法做到将全部东西融入到一次演讲中，也正因如此，我写了这本书。（你完全不需要为了理解这本书而特意去看我的演讲，不过视频仅有 20 分钟左右，而且 dotSwift 的剪辑工作非常棒，使我几乎摇身一变成了一位像样的讲者。不过我确信，你会爱上我迷人的匈牙利口音！）

本书面向的读者

从表面看，这本书的受众是那些想要自己实现集合类型的人，但切不可以知表不知里，Swift 中存在一些能使它更加独特的语言天赋，事实上本书的内容对于任何一个想要学习这些独特天赋的人来说都是十分有用的。无论是学会使用代数数据类型 (algebraic data type)，亦或是了解如何通过写时复制 (copy-on-write) 的引用计数存储来创建 *swifty* 的数据类型，都将有助于你在日常开发中成为更好的程序员。

本书假设你已经是有一定经验的 Swift 程序员。不过你完全不需要是 Swift 专家：如果你熟悉基本语法，而且已经写了数千行 Swift 代码，那么你一定能够顺利地理解本书。如果你需要快速上手 Swift，我会强烈推荐另一本 objc.io 的书给你，由 Chris Eidhof、Begemann 和 Airspeed Velocity 所著的 **Swift 进阶**。这本书可是说是 Apple 的 *The Swift Programming Language* 的接续，它对 Swift 的特性进行了更深入的挖掘，很好地解释了如何以一种符合语言特性 (或者说：*swifty*) 的方式来使用它们。

> 译者注：您也可以在 ObjC 中国的网站找到 Swift 进阶一书的中译版本。

本书中几乎所有代码都可以运行在支持 Swift 代码的任意平台上。除了在几个特例中，目前标准库和跨平台基础框架都不支持所需要使用的特性，所以我引入了平台特定的代码用以分别支持 Apple 和 GNU/Linux 平台。另外，本书的代码均在 Xcode 9 自带的 Swift 4 编译器中进行了测试。

书籍更新

随着时间的推进，我随时有可能发布新版本，或是为修复 bug，或是为补充资料，或是跟随 Swift 语言的进化而迭代。届时你将可以从你原来购买本书的页面下载更新。此外，你还可以在那里获取到不同格式的书籍：当前版本支持 EPUB、PDF 和 Xcode playground 三种格式。(只要登录购买该书所使用的账号，你将可以无限期免费下载。)

相关工作

我创建了一个 GitHub 仓库，你可以在那里找到书中提到的所有算法的完整源码。不过我只是将代码单纯地从书中提取出来，未做任何修饰，所以并不包含额外的信息。为了避免你想要尝试修改源码来进行测试却无法即兴而为之，我认为提供一个独立的源码包会是一个不错的选择。

你可以在自己的应用中随意使用任意来自上述仓库的代码，虽然诚实地说，很多时候这并不见得是一个好主意：为了配合本书，我将部分代码进行了一定程度的简化，因此并不一定满足生产代码所要求的质量。不过，我建议你看一看 **B 树** (BTree)，这是我精心为 Swift 量身打造的有序集合类型。我私以为这是本书中最先进的数据结构，而且代码实现完全满足生产代码的质量要求，在那里，你可以看到基于树实现的类似于标准库中 Array、Set 和 Dictionary 一样的集合类型，以及一个灵活的 BTree 数据类型，它可以用来对底层结构进行低层级访问。

Attabench 是我开发的 macOS 版的性能测试应用，用于为小段代码生成微型的性能测试图表。本书中实际使用的性能测试默认就包含在该应用中。我强烈建议你在自己的电脑中下载这个应用来实际试一试我所做过的测试。在这之后你也可以将自己的算法用性能测试图反映出来并进行种种探索。

联系作者

如果你发现任何错误，请在本书的 Github 仓库中提交一个 issue 来帮助我解决它。有其它任何类型的反馈，也随时欢迎在 Twitter 上联络我，账号是 *@lorentey*。如果你钟爱写邮件，发邮件到 *collections@objc.io* 也完全没问题。

如何阅读本书

我没有打破常规，所以本书的阅读顺序更倾向于从前至后。假设读者按照正序阅读，会发现书中有不少内容需要与前面的章节参照阅读。话虽然这么说，但按照自己的喜好的顺序来阅读也是可以的。不过答应我，这样做的时候就算感觉并不那么顺畅，也不要惆怅，好吗？

这本书包含大量源码。在 playground 版本的书籍中，几乎所有代码都可以编辑，而且你所做的修改会即时反映出来。你可以通过改动代码来亲身体会所讲述的内容 – 有时候最佳的理解方式正是看一看当你改变它时会发生什么。

比如说，Sequence 上有一个很有用的扩展方法，用于将所有元素乱序重排。那部分代码中有几个 FIXME 注释描述了代码实现存在的问题。不妨尝试修改代码来修复它们！

```
#if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
import Darwin // 为了支持 arc4random_uniform()
#elseif os(Linux)
import Glibc // 为了支持 random()
#endif

extension Sequence {
  public func shuffled() -> [Iterator.Element] {
    var contents = Array(self)
    for i in 0 ..< contents.count - 1 {
      #if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
        // FIXME: 数组元素数量超过 2^32 时会挂
        let j = i + Int(arc4random_uniform(UInt32(contents.count - i)))
      #elseif os(Linux)
        // FIXME: 这里存在模偏差（modulo bias）的问题。
        // 另外，应该通过调用 `srandom` 来为 `random` 配置随机种子。
        let j = i + random() % (contents.count - i)
      #endif
      if i != j {
        contents.swapAt(i, j)
      }
    }
    return contents
  }
```

```
}
```

为了说明一段代码被执行之后发生了什么，有时我会展示执行结果。作为例子，让我们来试着运行 shuffled，以证明每次运行都返回了新的随机顺序：

```
▶ (0 ..< 20).shuffled()
  [5, 12, 0, 1, 9, 16, 4, 8, 15, 19, 14, 11, 18, 7, 10, 6, 13, 3, 2, 17]
▶ (0 ..< 20).shuffled()
  [5, 10, 6, 11, 4, 2, 9, 8, 0, 18, 16, 7, 17, 14, 13, 19, 3, 1, 12, 15]
▶ (0 ..< 20).shuffled()
  [9, 15, 13, 17, 18, 4, 12, 14, 16, 2, 0, 5, 6, 19, 11, 7, 3, 1, 10, 8]
```

在 playground 版本的书籍中，所有输出结果都会被即时生成，因此你会在每次打开这一页时得到一组不同的乱序数字集。

致谢

如果没有读者们针对早期草稿给出的精彩绝伦的反馈，一定没有这本书的今天。除了我的读者们，我尤其还想要感谢 *Chris Eidhof*，他花了相当多的时间来审查早期的书稿，提出了很多详尽的反馈意见，使本书最终版得到了质的飞跃。

Ole Begemann 作为本书的技术审查者；没有问题能逃过他滴水不漏地审查。他绝妙的建议使得代码更加简明漂亮，而且他发现了很多就连我自己也从未意识到的令人惊叹的细节。

还因为有了 *Natalye Childress* 顶级的审校，我那笨拙且凌乱的句子们才得以转化成为一本真正用妥帖的英语写成的书。我绝对不是在夸大她的贡献；她几乎对每一个段落都做出了不少适当的调整。

当然了，书中也许尚存问题，但我绝不允许这一群很棒的人因此而被指责。如有不善，还请唯我是问。

最后我想感谢的是 *Floppy*，我七岁的比格犬：她总是耐心地听我描述纷繁复杂的技术问题，让我能够提供更好的问题解决方案。谢谢你，我的好孩子！

引言

1

集合类型是 Swift 语言的核心抽象概念之一。标准库中的主要集合类型包括：数组 (array)、集合 (set) 和字典 (dictionary)，从小脚本到大应用，它们被用在几乎所有的 Swift 程序中。Swift 程序员都熟悉它们的具体运作方式，而且它们的存在赋予了这门语言独特的个性。

当我们需要设计一个新的通用集合类型时，效仿标准库已经建立的先例不失为一个好办法。但是单纯遵循 Collection 协议的要求并不够，我们还需要再多做一些额外的工作来让它的行为与标准集合类型相匹配。本质上来说，就是要符合一些 **Swift 风格**的难以捉摸的性质，很难解释如何正确地做到这一切，但它们的缺席会让人痛不欲生。

写时复制 (copy-on-write) 值语义

不知道是否与你的想法不谋而合，我认为 Swift 集合类型中最重要的特性非**写时复制值语义**莫属。

从本质上来说，**值语义**在上下文中意味着每个变量都持有一个值，而且表现得**像是**拥有独立的复制，所以改变一个变量持有的值并不会修改其它变量的值：

```
var a = [2, 3, 4]
let b = a
a.insert(1, at: 0)
▶ a
  [1, 2, 3, 4]
▶ b
  [2, 3, 4]
```

为了实现值语义，上述代码需要在某些时候复制数组的底层存储，以允许两个数组实例拥有不同的元素。对于简单值类型 (像是 Int 或 CGPoint) 来说，整个值直接存储在一个变量中，当初始化一个新变量，或是将新值赋给已经存在的变量时，复制都会自动发生。

然而，将一个数组赋给新变量并**不会**发生底层存储的复制，这只会创建一个新的引用，它指向同一块在堆上分配的缓冲区，所以该操作将在常数时间内完成。直到指向共享存储的变量中有一个值被更改了 (例如：进行 insert 操作)，这时才会发生真正的复制。不过要注意的是，只有在改变时底层存储是共享的情况下，才会发生复制存储的操作。如果数组对它自身存储所持有的引用是唯一的，那么直接修改存储缓冲区也是安全的。

当我们说 Array 实现了**写时复制**优化时，我们本质上是在对其操作性能进行一系列相关的保证，从而使它们表现得就像上面描述的一样。

(要注意的是，完整的值语义通常被认为是由一些名字很可怕的抽象概念组成，就像是**引用透明** (referential transparency)、**外延性** (extensionality) 和**确定性** (definiteness)。在某种程度上，Swift 的标准集合违反了每一条。比如说，就算两个集合包含完全相同的元素，一个集合的索引在另一个集合中也并不一定有效。因此，Swift 的集合并不是**完全**引用透明的。)

SortedSet 协议

在开始之前，我们首先需要确定一个想要解决的课题。目前标准库中缺少一个非常常用的数据结构：有序集合 (sorted set) 类型，这是一个类似 Set 的集合类型，但是要求元素是 Comparable (可比较的)，而非 Hashable (可哈希的)，此外，它的元素保持升序排列。接下来，让我们卯足火力来实现一个这样的集合类型吧！

这本书将始终围绕有序集合问题进行，对于用多种方法构建数据结构来说，无疑这会是一个很好的示范。之后我们将会创造一些独立的解决方案，并 (举例) 说明一些有趣的 Swift 编码技术。

现在，我们来起草一份想要实现的 API 协议作为开始。理想情况下，我们希望创建遵循下述协议的具体类型：

```
public protocol SortedSet: BidirectionalCollection, SetAlgebra {
  associatedtype Element: Comparable
}
```

有序集合的核心是将多个元素按一定顺序放置，所以实现 BidirectionalCollection 是一个合情合理的需求，这允许从前至后遍历，也允许自后往前遍历。

SetAlgebra 包含所有的常规集合操作，像是 union(_:)、isSuperset(of:)、insert(_:) 和 remove(_:)，以及创建空集合或者包含特定内容的集合的初始化方法。如果我们志在实现产品级的有序集合，那么毫无疑问，没有理由不完整实现该协议。然而，为了让这本书在可控范围内，我们将只实现 SetAlgebra 协议中很小的一部分，包括 contains 和 insert 两个方法，再加上用于创建空集合的无参初始化方法：

```
public protocol SortedSet: BidirectionalCollection, CustomStringConvertible,
↪  CustomPlaygroundQuickLookable
```

```
  where Element: Comparable
{
  init()
  func contains(_ element: Element) -> Bool
  mutating func insert(_ newElement: Element) -> (inserted: Bool, memberAfterInsert:
  ↪  Element)
}
```

作为放弃完整实现 SetAlgebra 的交换，我们添加了 CustomStringConvertible 和 CustomPlaygroundQuickLookable；这样一来，当我们想要在示例代码和 playground 中显示有序集合的内容时，能够稍微得心应手一些。

BidirectionalCollection 协议有大约 30 项要求 (像是 startIndex、index(after:)、map 和 lazy)，好消息是，它们中的大多数都有默认实现。如果想要使一个类型满足该协议，最小情况下我们需要实现其中的五个成员：startIndex、endIndex、subscript、index(after:) 和 index(before:)。在本书中，我们将稍微更进一步，实现专门的 forEach 和 count。当能获益很多的时候，我们也会为 formIndex(after:) 和 formIndex(before:) 添加自定义的实现。对于其他的多数部分，有时通过写专门进行针对代码的可以让它们工作得更快，不过我们还是将使用标准库中默认的实现。

语义要求

通常，实现一个 Swift 协议意味着不仅要遵循它的明确要求，大多数协议还具有一系列在类型系统中无法表达的附加语义要求。这些要求需要被单独写成文档。SortedSet 协议也不例外，我们期望所有实现都能满足下述的六个性质。

1. **有序：**集合类型中的元素需要时刻保持已排序状态。具体一点说就是：如果在实现了 SortedSet 的 set 中，i 和 j 都是有效的下标索引，那么 i < j 必须与 set[i] < set[j] 等效。(这个例子也暗示了我们，集合没有重复元素。)
2. **值语义：**通过一个变量更改 SortedSet 类型的实例时，必须不能影响同类型的任意其他变量。这也就是说，我们需要遵循：类型必须表现得**像是**每个变量都拥有自己的独一无二的值，完全独立于其它所有变量。
3. **写时复制：**复制一个 SortedSet 值到新变量的复杂度应该是 $O(1)$。存储可能在不同的 SortedSet 值之间部分或完全共享。当需要满足值语义时，所有更改都必须先检查共享

存储，并在合适的时机创建新的复制。因此，当存储被共享的时候，一旦发生改变可能需要较长的时间才能完成整个处理。

4. **特定索引：**索引和特定的 SortedSet 实例相关联，它们只保证对于这个特定的实例和它的不可变直接复制是有效的。即使 a 和 b 是包含完全相同元素的同一类型的 SortedSet 实例，a 的索引在 b 中也未必有效。(通常在技术上，这种对真的值语义的放宽是一种无奈之举，似乎很难避免。)

5. **索引失效：**任何 SortedSet 的改变都**可能**导致所有已经存在的索引失效，包括 startIndex 和 endIndex。对于具体实现来说，让所有的索引失效并不总是**必要的**，但是想这么做也没问题。(这一点并不能算是要求，因为这从根本上来说不可能被违背。这只是一个提醒，让我们铭记在心，我们需要假定集合类型的索引非常脆弱，需要谨慎进行处理。)

注意，如果你忘记实现任意一个要求，编译器并不会提醒你。但是实现它们是至关重要的，只有这样，使用有序集合的一般代码才能有稳定的行为。

假如我们正在实现一个现实可用，满足生产要求的有序集合，我们完全没有必要实现 SortedSet 协议，而只需简单地定义一个直接实现了所有要求的单一类型即可。然而，我们将编写不止一个有序集合，因此有一个规定了所有要求的协议再好不过了，而且我们可以基于它定义通用扩展。

虽然我们还没有具体实现 SortedSet，但是果断先来定义一个通用扩展又何尝不是一个激动人心的选择呢！

打印有序集合

提供一个 description 的默认实现能够让我们免去今后设置输出格式的麻烦。由于所有的有序集都是集合类型，我们完全可以使用标准集合类型的方法来打印它们，就像标准库的数组和集合一样，将元素用逗号分隔，并用括号括起来：

```
extension SortedSet {
  public var description: String {
    let contents = self.lazy.map { "\($0)" }.joined(separator: ", ")
    return "[\(contents)]"
  }
}
```

此外，为 customPlaygroundQuickLook 创建一个默认实现也很有价值，这样我们的集合类型在 playground 中的输出也能稍微优美一些。一眼看上去，默认的 Quick Look 视图很难理解，所以我使用属性字符串 (attributed string)，将 description 的字体设置为等宽字体，并以此来代替原来的视图。

```
#if os(iOS)
import UIKit

extension PlaygroundQuickLook {
  public static func monospacedText(_ string: String) -> PlaygroundQuickLook {
    let text = NSMutableAttributedString(string: string)
    let range = NSRange(location: 0, length: text.length)
    let style = NSParagraphStyle.default.mutableCopy() as! NSMutableParagraphStyle
    style.lineSpacing = 0
    style.alignment = .left
    style.maximumLineHeight = 17
    text.addAttribute(.font, value: UIFont(name: "Menlo", size: 13)!, range: range)
    text.addAttribute(.paragraphStyle, value: style, range: range)
    return PlaygroundQuickLook.attributedString(text)
  }
}
#endif

extension SortedSet {
  public var customPlaygroundQuickLook: PlaygroundQuickLook {
    #if os(iOS)
      return .monospacedText(String(describing: self))
    #else
      return .text(String(describing: self))
    #endif
  }
}
```

有序数组
(Sorted Arrays)

2

想要实现 SortedSet，也许最简单的方法是将集合的元素存储在一个数组中。这引出了一个像下面这样的简单结构的定义：

```
public struct SortedArray<Element: Comparable>: SortedSet {
  fileprivate var storage: [Element] = []

  public init() {}
}
```

为了满足协议的要求，我们会时刻保持 storage 数组处于已排序的状态，故此，命其名曰 SortedArray。

二分查找

为了实现 insert 和 contains，我们需要一个方法，给定一个元素，该方法返回该元素在数组中应当放置的位置。

如何快速实现这样一个方法呢？首先我们需要实现**二分查找算法**。这个算法的工作原理是，将数组一分为二，舍弃不包含我们正在查找的元素的那一半，将这个过程循环往复，直到减少到只有一个元素为止。下面是 Swift 中实现该算法的方法之一：

```
extension SortedArray {
  func index(for element: Element) -> Int {
    var start = 0
    var end = storage.count
    while start < end {
      let middle = start + (end - start) / 2
      if element > storage[middle] {
        start = middle + 1
      }
      else {
        end = middle
      }
    }
    return start
```

```
  }
}
```

值得注意的是，即使我们将集合的元素数量加倍，上述循环也仅仅只需要多进行一次迭代。这可以说是代价相当低了！人们常常说二分查找具有**对数复杂度** (logarithmic complexity)，具体来说就是：它的运行时间与数据规模大小大致呈对数比。(用大 O 符号来描述则是：$O(\log n)$。)

二分查找是一个巧妙的算法，看似简单，实则暗藏玄机，正确地实现它并不是一件容易的事情。二分查找包含许多索引计算，以至于发生错误的几率并不低，像是差一错误 (off-by-one errors)、溢出问题等等。举个例子：我们运用了表达式 start + (end - start) / 2 来计算中间索引，这看起来似乎有些歪门邪道；通常会更直观地写为 (start + end) / 2。然而，这两个表达式并不总是能够获得相同结果，因为第二个版本的表达式包含的加法运算可能会在集合类型元素数量过多时发生溢出，从而导致运行时错误。

我希望有朝一日二分查找能被纳入 Swift 标准库。在此之前，如果什么时候你需要实现二分查找，务必找一本好的算法书籍作为参考。(尽管我认为这本书也会有一些帮助。) 还有，不要忘记测试你的代码，有时候即使是书中的代码也有 bug！我发现覆盖率 100% 的单元测试能帮助我捕获大多数错误。

我们的 index(for:) 函数所做的事情与 Collection 的标准 index(of:) 方法很相似，不同的是，即使要查找的元素并不存在于当前集合，我们的版本也还是能返回一个有效索引。这个细微但是十分重要的不同点能够让 index(for:) 在插入操作中也相当好用。

查找方法

提到 index(of:)，我认为借助 index(for:) 来定义它也不失为一个好主意，这样一来它也可以用到更好的算法：

```
extension SortedArray {
  public func index(of element: Element) -> Int? {
    let index = self.index(for: element)
    guard index < count, storage[index] == element else { return nil }
    return index
  }
}
```

Collection 的默认查找算法的原理是：执行一个线性查找来遍历所有元素，直到找到目标或是到达末尾为止。经过我们专门优化后的版本要快得**多的多**。

检验元素与集合类型的所属关系所需要的代码会稍微少一点，因为我们只需要知道元素是否存在：

```
extension SortedArray {
  public func contains(_ element: Element) -> Bool {
    let index = self.index(for: element)
    return index < count && storage[index] == element
  }
}
```

实现 forEach 更加容易，因为我们可以直接将这个调用传递给我们的存储数组。数组已经排序，因此这个方法将会以正确的顺序访问元素：

```
extension SortedArray {
  public func forEach(_ body: (Element) throws -> Void) rethrows {
    try storage.forEach(body)
  }
}
```

到现在我们已经实现了几个方法，不妨回过头看一看其他 Sequence 和 Collection 的成员，值得开心的是，它们也受益于专门的实现。比如说，由 Comparable 元素组成的序列有一个 sorted() 方法，返回一个包含该序列所有元素的有序数组。对于 SortedArray，简单地返回 storage 就可以实现：

```
extension SortedArray {
  public func sorted() -> [Element] {
    return storage
  }
}
```

插入

向有序集合中插入一个新元素的流程是：首先用 index(for:) 找到它相应的索引，然后检查这个元素是否已经存在。为了维护 SortedSet 不能包含重复元素特性，我们只向 storage 插入目前不存在的元素：

```
extension SortedArray {
  @discardableResult
  public mutating func insert(_ newElement: Element) -> (inserted: Bool,
  ↪   memberAfterInsert: Element)
  {
    let index = self.index(for: newElement)
    if index < count && storage[index] == newElement {
      return (false, storage[index])
    }
    storage.insert(newElement, at: index)
    return (true, newElement)
  }
}
```

实现集合类型

下一步，让我们来实现 BidirectionalCollection。因为我们将所有东西都存储到了一个单一数组中，所以最简单的实现方法是在 SortedArray 和它的 storage 之间共享索引。这样一来，我们可以将大多数集合类型的方法直接传递给 storage 数组，从而大幅度简化我们的实现。

Array 实现的不止是 BidirectionalCollection，实际是有着相同 API 接口但语义要求更严格的 RandomAccessCollection。RandomAccessCollection 要求高效的索引计算，因为我们必须任何时候都能够将索引进行任意数量的偏移，以及测算任意两个索引之间的距离。

一个事实是，我们无论如何都会向 storage 传递各种调用，所以在 SortedArray 上实现相同的协议是一件有意义的事情：

```
extension SortedArray: RandomAccessCollection {
  public typealias Indices = CountableRange<Int>
```

```
  public var startIndex: Int { return storage.startIndex }
  public var endIndex: Int { return storage.endIndex }

  public subscript(index: Int) -> Element { return storage[index] }
}
```

这样我们就完成了 SortedSet 协议的实现。太棒了！

例子

让我们来检验一下是否一切正常：

```
var set = SortedArray<Int>()
for i in (0 ..< 22).shuffled() {
  set.insert(2 * i)
}
▶ set
  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
↪  42]

▶ set.contains(42)
  true

▶ set.contains(13)
  false
```

看起来不错。但是我们的新集合类型是否具有值语义？

```
let copy = set
set.insert(13)

▶ set.contains(13)
  true
```

```
▶ copy.contains(13)
  false
```

看起来答案是肯定的！我们并没有做任何工作来实现值语义；凭借 SortedArray 是一个由单一数组构成的结构体这个仅有的事实，我们得到了前面结果。值语义是一个组合性质，若结构体中的存储属性全都具有值语义，它的行为也会自动表现得一致。

性能

当我们谈论一个算法的性能时，我们常用所谓的**大 O 符号**来描述执行时间受输入元素个数的影响所发生的改变，记为：$O(1)$、$O(n)$、$O(n^2)$、$O(\log n)$、$O(n \log n)$ 等。这个符号在数学上有明确的定义，不过你不需要太关注，理解我们在为算法**增长率** (growth rate) 分类时使用这个符号作为简写就足够了。当输入元素个数倍增时，一个 $O(n)$ 的算法会花费不超过两倍的时间，但是一个 $O(n^2)$ 的算法可能比从前慢四倍，同时一个 $O(1)$ 的算法的执行时间并不大会受输入影响。

我们可以基于数学来分析我们的算法，合理地推导出渐进复杂度估计值。分析能为我们提供关于性能的有用指标，但它不是绝对的；就其本质而言，由于依赖简化的模型，与真实世界中的实际硬件的行为既有可能相匹配，也有可能存在差池。

为了了解我们的 SortedSet 的真实性能，运行一些性能测试是个好办法。例如，下述代码可以对四个 SortedArray 上的基础操作进行微型性能测试，它们分别是：insert、contains、forEach 和用 for 语句实现的迭代：

```
func benchmark(count: Int, measure: (String, () -> Void) -> Void) {
  var set = SortedArray<Int>()
  let input = (0 ..< count).shuffled()
  measure("SortedArray.insert") {
    for value in input {
      set.insert(value)
    }
  }

  let lookups = (0 ..< count).shuffled()
  measure("SortedArray.contains") {
    for element in lookups {
```

```
        guard set.contains(element) else { fatalError() }
      }
    }

    measure("SortedArray.forEach") {
      var i = 0
      set.forEach { element in
        guard element == i else { fatalError() }
        i += 1
      }
      guard i == input.count else { fatalError() }
    }

    measure("SortedArray.for-in") {
      var i = 0
      for element in set {
        guard element == i else { fatalError() }
        i += 1
      }
      guard i == input.count else { fatalError() }
    }
}
```

measure 参数是测量其闭包执行时间的函数，第一个参数表示它的名字。驱动 benchmark 函数的一个简单方法是在不同元素个数的循环中调用它，并打印测量结果：

```
for size in (0 ..< 20).map({ 1 << $0 }) {
  benchmark(size: size) { name, body in
    let start = Date()
    body()
    let end = Date()
    print("\(name), \(size), \(end.timeIntervalSince(start))")
  }
}
```

这是我实际用来画出下面图表时所使用的 Attabench 性能测试框架的简化版。真实的代码中含有更多的测试模板之类的东西，不过实际的测量方式 (measure 闭包中的代码) 并无二致。

绘制我们的性能测试结果，得到 图 2.1。注意，在这个图表中，我们对两个坐标轴都使用了对数标度 (logarithmic scales)，这意味着：向右移动一个刻度，输入值的数量翻一倍；向上移动一条水平线，执行时间增长为十倍。

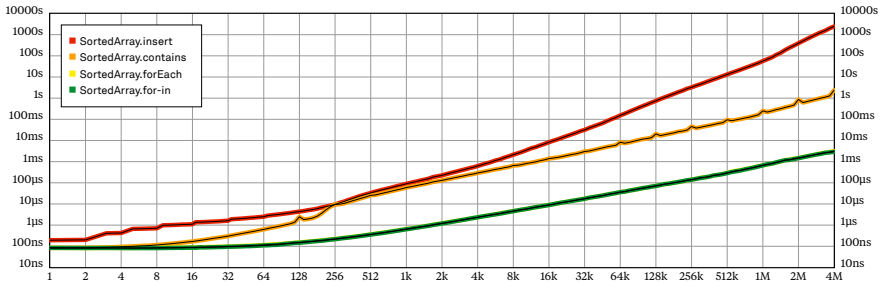


图 2.1: SortedArray 操作的性能测试结果，在双对数坐标系上描画输入值的元素个数和总体执行时间。

双对数坐标系非常适合用来表示性能测试结果。不仅可以无压力地在单一图表上表示跨度巨大的数据，而且有效避免了小值被埋没在大值的世界里。在这个例子中，我们可以很容易地比较元素数量从一增加到四百万的执行时间，尽管它们之间的差异达到了惊人的 22 个二的幂次数量级！

此外，双对数坐标系让我们能够简单地估计一个算法展现的实际复杂度。如果性能测试中某部分是一条直线，那么输入元素个数和执行时间之间的关系近似于一个简单多项式的倍数，如 n、n^2 甚至是 $\sqrt{n}$。指数与直线的斜率相关联，n^2 的斜率是 n 的两倍。在有了一些亲身实践之后，你会对发生频率最高的关系一目了然，完全没有必要进行复杂的分析。

在我们的例子中，单纯地迭代数组中的所有元素应该会花费 $O(n)$ 的时间，这在我们的图中也得到了证实。Array.forEach 和 for-in 循环的时间成本几乎相同，而且在初始热身周期之后，它们都变成了直线。横坐标向右移动三个单位多一点，纵坐标就向上移动一个单位，相当于 $2^{3.3} \approx 10$，这证明了一个简单的线性关系。

再来看一看 SortedArray.insert 的图，我们会发现元素数量约为 4,000 时它逐渐变化成为一条直线，斜率大致为 SortedArray.forEach 斜率的两倍，由此可以推断插入的执行时间是输入元素数量的二次函数。我们从理论上进行的推测是：每次向已排序数组插入一个随机元素的时候，需要将 (平均) 一半的既有元素向右移动一位来给插入元素腾出位置。因此插入是一个线性操作，n 个插入操作需要花费 $O(n^2)$。很幸运，图表走势与我们的预期相吻合。

SortedArray.contains 进行 n 次二分查找, 每次花费 $O(\log n)$ 的时间，因此它应该是一个 $O(n \log n)$ 的函数。这很难从 图 2.1 中看出来，但是如果你离近了仔细看，便可以验证我们的推测：contains 的曲线几乎平行于 forEach 的曲线，只是稍微向上偏离，但它不是一条完美的直线。你可以将一张纸的边缘放到 contains 图的旁边来进行验证，它弯弯曲曲远离了纸的直边，反映了一种超线性 (superlinear) 关系。

为了突出 $O(n)$ 和 $O(n \log n)$ 之间的差异，一个不错的方案是：用输入元素个数除以执行时间，并将结果反映在图表中来展示花费在一个元素上的平均执行时间。(我喜欢把这种类型的图称为**平摊图** (amortized chart)。我不确定在上下文中使用**平摊**合不合适，但是这个词语很容易给人留下深刻的印象！) 这个除法运算排除了斜率始终不变的 $O(n)$，使得我们可以简单地区分线性因子和对数因子。图 2.2 展示的是 SortedArray 的平摊图。你会发现，现在 contains 有一个明显 (但是细微) 的向上趋势，而 forEach 的尾部趋于完全水平。

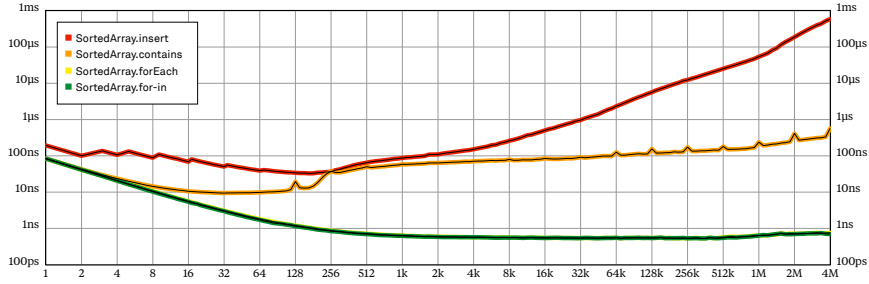


图 2.2: SortedArray 操作的性能测试结果，在双对数坐标系上描画输入值的元素个数和单次操作的平均执行时间。

contains 的曲线带来了两个意料之外的事实。其一是：在元素个数为 2 的幂次方时，会出现一个明显的尖峰。这是因为在二分查找和运行性能测试的 MacBook 的二级 (L2) 缓存架构之间存在一个有意思的相互作用。缓存被分为一些 64 字节的**行** (line)，其中每一部分都可能持有来自一系列特定物理地址的内存中的内容。由于一个不幸的巧合，如果存储大小接近于 2 的幂次方时，二分查找算法的连续查找操作可能会落入相同的 L2 缓存行，从而迅速耗尽它的容量，其它行却处于未使用状态。这个现象被称为**缓存行别名** (cache line aliasing)，它会导致一个极具戏剧性的性能衰退：contains 峰值耗费的执行时间约为相邻元素个数耗时的两倍。

消除这些尖峰的一种方法是改用**三分查找** (ternary search)，每次迭代时将缓存等分为**三个**部分。还有一种更简单的解决方案，选择一个略微偏离中心的位置作为中心索引来扰乱二分查找。如果选择这个方案，我们只需要在 index(for:) 的实现中修改一行即可，在中心索引上添加一个额外的小偏移量：

```
let middle = start + (end - start) / 2 + (end - start) >> 6
```

这样的话，中间索引将落在两个端点的 $33/64$ 处，足以避免缓存行别名现象。不幸的是，代码变得稍微复杂了一点，相较于二分查找，这些偏离正中的中心索引通常会导致存储查找次数小幅增加。这样看来，消除 2 的幂次方的尖峰所需付出的代价是总体上的衰退，在图表中也得到了证明，如 图 2.3 中所示。

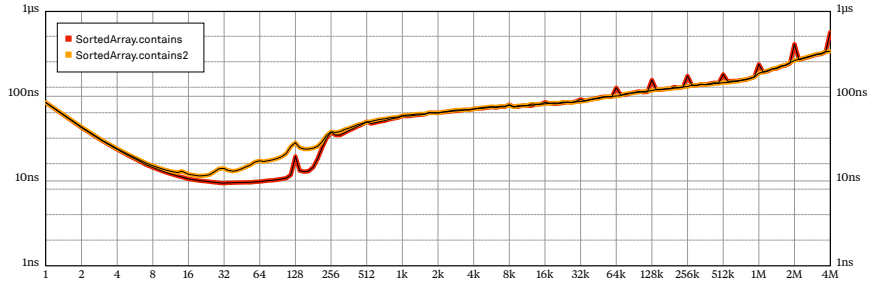


图 2.3: 比较二分查找 (contains) 和使用中心索引偏移来避免缓存行别名的版本 (contains2) 的性能。

还记得上文说的 contains 曲线带来了两个意料之外的事实吗？其二是：在 64,000 个元素及之后，曲线出现轻微上升 (斜率变大)。(如果你仔细观察，你可能会察觉到从大概一百万个元素开始，insert 发生了一个虽然不太明显，但是很类似的衰退。) 对于这种规模的元素个数，我的 MacBook 的虚拟内存子系统无法保持 CPU 的地址缓存 (也叫做页表缓存 (Translation Lookaside Buffer)，简称 TLB) 中 storage 数组的所有分页的物理地址。再加上 contains 的性能测试进行的是随机查找，它毫无规律的访问模式导致了 TLB 频繁发生缓存未命中，很大程度上增加了内存访问的成本。另外，随着存储数组的元素个数越来越多，它的绝对大小超过 L1 和 L2 缓存，那些缓存未命中造成了大量附加延迟。

所以在最后，看起来在一个足够大的连续缓冲区进行随机内存访问要花费 $O(\log n)$ 的时间，而远非 $O(1)$，所以我们的二分查找的渐进执行时间实际更像是 $O(\log n \log n)$，而非我们通常认为的 $O(\log n)$。这结果是不是很有趣？(如果我们从性能测试的代码中将在 lookups 数组上调用的用来随机打乱数组的 shuffled 方法移除，衰退便会烟消云散。试试看！)

另一方面，对于元素个数少的情况，contains 的曲线与 insert 其实非常接近。一部分原因可以归结为对数刻度的副作用，在它们接近的位置，contains 仍然比 insert 快了近 80%。但是 insert 曲线在大约 1,000 个元素时平坦得令人吃惊，似乎当有序数组足够小的时候，插入一个新元素所耗费的时间与数组大小无关。(我认为这是因为在元素个数处于这些区间的时候，整个数组可以完全放入 CPU 的 L1 缓存。)

数组元素足够少的时候，SortedArray.insert 似乎快的难以置信。目前我们可以把这件事视作无关紧要的有趣的假说。但是务必把它牢记在心，因为我们会在本书后面的部分对这个事实进行严肃的讨论。

将 NSOrderedSet Swift 化

3

Foundation 框架包含一个名为 NSOrderedSet 的类。它首次于 2012 年登场，与 iOS 5 和 OS X 10.7 Lion 一同诞生，是一个很年轻的类。NSOrderedSet 被添加到 Foundation 中的主要目的是支持 Core Data 中的有序关系。它就像 NSArray 和 NSSet 的合成体一样，同时实现了两个类的 API。正因如此，它提供了 NSSet 中复杂度仅为 $O(1)$ 的超快成员关系检查和 NSArray 的 $O(1)$ 复杂度的随机访问索引。作为折中，它继承了 NSArray 的 $O(n)$ 的插入。由于 NSOrderedSet (算) 是通过封装 NSSet 和 NSArray 实现的，所以相比两者中任意一个，它的内存消耗都要更高一些。

NSOrderedSet 目前还尚未被桥接到 Swift，在这个前提下，尝试为 Objective-C 类定义简单的封装，使其更接近 Swift 的世界，看起来是一个不错的主题。

尽管 NSOrderedSet 是一个很酷的名字，但这与我们的用例并不是十分匹配。NSOrderedSet 的元素的确是有顺序的，不过它并没有强制要求特定的有序关系，你可以以任何喜欢的顺序来进行元素插入，NSOrderedSet 会像一个数组一样为你记住这一切。“ordered” 和 “sorted” 之间的区别在于是否有一个预定义的顺序，这也是为什么 NSOrderedSet 并不能被称为 NSSortedSet 的原因。这么做最根本的目的是让查找操作的速度足够快，不过它使用的实现方法是哈希而非比较。(Foundation 中不存在与 Comparable 协议等效的东西；NSObject 只提供 Equatable (可判等) 和 Hashable (可哈希) 功能。)

但是只要 NSOrderedSet 的元素实现了 Comparable 的话，我们就可以做到保持元素按大小排列，而不仅仅是按插入顺序排列。很明显，对 NSOrderedSet 而言这并不算是理想的使用方式，但是我们确实是可以做到这一点的。接下来就让我们引入 Foundation，开始着手于将 NSOrderedSet 锤炼为 SortedSet：

```
import Foundation
```

不过马上我们就遇到了几个大问题。

第一，NSOrderedSet 是一个类，所以它的实例是引用类型。而我们想要让有序集合具有值语义。

第二，NSOrderedSet 是一个混合类型序列，它接受 Any 类型作为成员。实现 SortedSet 时我们依然可以设置它的 Element 类型为 Any，而不是将其作为泛型参数，但是感觉这和我们想要的解决方案还有些差距。我们真正期待的是一个泛型的同质集合类型，它可以通过类型参数来指定其中的元素类型。

基于上述原因，我们不能够只通过扩展 NSOrderedSet 来实现我们的协议。取而代之，我们将会定义一个泛型的封装结构体，它的内部使用 NSOrderedSet 的实例作为存储。这种方法类似

于 Swift 标准库为了将 NSArray、NSSet 和 NSDictionary 实例桥接到 Swift 的 Array、Set 和 Dictionary 值时所做的工作。这样看来，我们似乎步入了正轨。

我们应该给结构体起个什么名字呢？NSSortedSet 这个想法浮现上来，而且在技术上这是可行的，同时 Swift 限定的构造 (现在和将来都) 并不依赖于使用前缀来解决命名冲突。但站在另一方面来看，对于开发者而言，NS 依然暗示着 **Apple 提供**，所以冒然使用显得很不礼貌，还极容易混淆。我们不妨换个思路，将我们的结构体命名为 OrderedSet。(虽然这个名字也不太正确，但至少像是一个基本数据结构的名字。)

```
public struct OrderedSet<Element: Comparable>: SortedSet {
  fileprivate var storage = NSMutableOrderedSet()
}
```

我们希望能够修改存储，所以需要将它声明为一个 NSMutableOrderedSet 的实例，NSMutableOrderedSet 是 NSOrderedSet 的可变子类。

查找元素

现在我们有一个数据结构的空壳。让我们用内容填满它，首先从 forEach 和 contains 这两个查找方法开始。

NSOrderedSet 实现了 Sequence，所以它已经有了一个 forEach 方法。假如元素能够保持正确的顺序，我们可以简单地将 forEach 的调用传递给 storage。然而，我们需要先手动将 NSOrderedSet 提供的值向下转换 (downcast) 为正确类型：

```
extension OrderedSet {
  public func forEach(_ body: (Element) -> Void) {
    storage.forEach { body($0 as! Element) }
  }
}
```

OrderedSet 对自身存储具有完全控制权，因此它可以保证存储中永远不会包含除了 Element 以外的任何类型的东西。这确保了向下强制类型转换一定会成功。不过说实话这不太优雅！

NSOrderedSet 恰好也为 contains 提供了实现，而且对于我们的用例来说似乎是完美的。因为不需要显式类型转换，它显得比 forEach 更易于使用：

```
extension OrderedSet {
  public func contains(_ element: Element) -> Bool {
    return storage.contains(element)  // BUG!
  }
}
```

编译上面的代码没有任何警告，当 Element 是 Int 或 String 的时候，它表现得一切正常。但是，正如我们已经提到过的，NSOrderedSet 使用了 NSObject 的哈希 API 来加速元素查找。而我们并未要求 Element 实现 Hashable！这凭什么可以正常工作呢？

当我们像上面的 storage.contains 中做的那样，将一个 Swift 值类型提供给一个接受 Objective-C 对象的方法时，编译器会为此生成一个私有的 NSObject 子类，并将值装箱 (box) 到其中。一定要记住 NSObject 有内建的哈希 API；你不可能有一个不支持 hash 的 NSObject 实例。因此，这些自动生成的桥接类也必然有与 isEqual(:) 一致的 hash 实现。

如果 Element 正好实现了 Hashable，那么 Swift 可以直接在桥接类中使用原类型自己的 == 和 hashValue 实现，这样一来，在 Objective-C 和 Swift 中取得 Element 的值的哈希值就是同样的方法了，而且两者都表现得很完美。

然而，如果 Element 没有实现 hashValue，那么桥接类就只有唯一的选择，那就是使用 NSObject 默认实现的 hash 和 isEqual(_:)。由于没有其它可用信息，它们都将基于实例的标志符 (即物理地址)，而对于被装箱的值类型而言，这是完全随机的。所以两个不同的桥接实例即使持有两个完全相同的值，也不会被认为相等 (或是返回相同的 hash)。

上面的这一切最终使 contains 可以通过编译，但是它却有一个致命的 bug：如果 Element 并未实现 Hashable，则查找总会返回 false。哎呀，糟糕了！

亲爱的，这是一个教训：在 Swift 中使用 Objective-C 的 API 时一定要非常非常小心。将 Swift 值自动桥接到 NSObject 实例确实很便利，但是也存在不易察觉的陷阱。关于这个问题，代码中不会有任何明确的警告：没有感叹号，没有显示转换，什么都没有。

现在我们知道了，在我们的例子中并不能够依赖 NSOrderedSet 的查找方法。所以我们不得不寻找其他 API 来查找元素。谢天谢地，NSOrderedSet 已经包含了另一个查找元素的方法，它依据比较函数的结果对一系列元素进行排序：

```
class NSOrderedSet: NSObject { // 在 Foundation 中
  ...
  func index(of object: Any, inSortedRange range: NSRange, options:
  ↪  NSBinarySearchingOptions = [], usingComparator: (Any, Any) -> ComparisonResult)
  ↪  -> Int
  ...
}
```

我推测这是二分查找某种形式的实现，所以它应该足够快。我们的元素可以根据它们的 Comparable 特性进行排序，因此我们可以使用 Swift 的 < 和 > 操作符来定义一个适合的比较器函数：

```
extension OrderedSet {
  fileprivate static func compare(_ a: Any, _ b: Any) -> ComparisonResult
  {
    let a = a as! Element, b = b as! Element
    if a < b { return .orderedAscending }
    if a > b { return .orderedDescending }
    return .orderedSame
  }
}
```

我们可以使用这个比较器来定义一个获取特定元素索引的方法。这正好是 Collection 的 index(of:) 方法应当做的，所以需要确保我们的定义让默认实现更加优雅：

```
extension OrderedSet {
  public func index(of element: Element) -> Int? {
    let index = storage.index(
      of: element,
      inSortedRange: NSRange(0 ..< storage.count),
      usingComparator: OrderedSet.compare)
    return index == NSNotFound ? nil : index
  }
}
```

我们有这个函数以后，对 contains 的改造就可以降低到一个很小的范围内：

```
extension OrderedSet {
  public func contains(_ element: Element) -> Bool {
    return index(of: element) != nil
  }
}
```

不知道你感觉如何，我发现事情比我预想的要更复杂一些。在如何将值桥接到 Objective-C 的问题上，细节**有时**会带来深远的影响，这可能会以难以察觉却致命的方法破坏我们的代码。如果我们不知道这些玄机的话，很难不经历意料之外的痛苦。

NSOrderedSet 的 contains 实现特别快，这是它的一个旗舰特性，所以不能够使用 contains 这件事就显得更加悲伤了。但是天无绝人之路！考虑到某些类型下 NSOrderedSet.contains 可能错误地返回 false，但如果值不是确实存在于集合里，它也绝不会返回 true。所以，我们可以写一个新版本的 OrderedSet.contains，依然在其中调用原版本方法，但省去了一部分场景下的二分查找需求：

```
extension OrderedSet {
  public func contains2(_ element: Element) -> Bool {
    return storage.contains(element) || index(of: element) != nil
  }
}
```

对于实现了 Hashable 的元素而言，这个版本返回 true 的速度比 index(of:) 更快。不过，遇到值并非集合的成员，或者类型不是可哈希的这两种情况时，处理速度会略微慢一点点。

实现 Collection

NSOrderedSet 只遵循 Sequence，而不遵循 Collection。(这不是什么独特的巧合；它有名的小伙伴 NSArray 和 NSSet 也一样。) 不过，NSOrderedSet 提供了一些基于整数的索引方法，我们可以使用它们在 OrderedSet 中实现 RandomAccessCollection。

```
extension OrderedSet: RandomAccessCollection {
  public typealias Index = Int
  public typealias Indices = CountableRange<Int>
```

```
    public var startIndex: Int { return 0 }
    public var endIndex: Int { return storage.count }
    public subscript(i: Int) -> Element { return storage[i] as! Element }
}
```

事实证明，这出乎意料的简单。

保证值语义

SortedSet 要求值语义，这意味着每个包含有序集合的变量都需要表现地像是持有自身的值的单独复制，完全与所有其它变量独立。

这次我们不会再 “免费” 得到值语义了！我们的 OrderedSet 结构体包含一个指向类实例的引用，所以拷贝一个 OrderedSet 的值到另一个变量只会增加存储对象的引用计数。

这意味着两个 OrderedSet 的变量可能很容易共享相同的存储：

```
var a = OrderedSet()
var b = a
```

上述代码的执行结果如图所示 图 3.1。

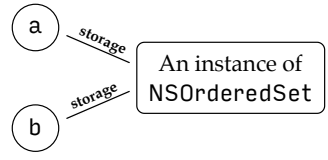


图 3.1: 两个 **OrderedSet** 的值共享指向同一个存储对象的引用。

虽然存储可能被共享，但像 insert 这类可变方法一定只能修改调用它的变量所持有的集合实例。实现方法之一是：在我们修改它之前，总是复制一个全新的 storage。不过，这样做也太浪费了，很多时候我们的 OrderedSet 的值持有对其存储的唯一引用，在这种情况下，即使不做复制而是直接进行修改也是安全的。

Swift 标准库提供了一个名为 isKnownUniquelyReferenced 的函数，可以调用它来判断一个指向对象的特定引用是否唯一。如果返回 true，那我们就知道没有其它值持有该对象的引用，所以直接修改它是安全的。

(务必注意，这个函数只关注强引用；并不计算弱引用和无主 (unowned) 引用。因此我们不可能**真正地**明察每一种引用持有的情况。还好，在我们的例子中这不是问题，由于 storage 是一个私有属性，只有 OrderedSet 内部的代码才可以访问它，我们也绝不会创建 “隐式” 引用。不计算弱引用和无主引用是故意而为，并非偶然的疏忽；这样一来，更复杂的集合类型的索引就可以在某些情况下 (比如将一个元素从特定索引移除时)，不进行强制写时复制，也能包含对存储的引用。我们将会在本书后面的章节中见到像这样的索引定义的例子。)

然而，还有一个很重要的问题：isKnownUniquelyReferenced 从来不会为 NSObject 的子类返回 true，因为子类有它们自己的引用计数实现，所以无法保证总是能返回一个正确的结果。毫无疑问，NSOrderedSet 也是 NSObject 的一个子类，这么看来我们完蛋了，这简直让人绝望！

噢，等等！如果我们将 OrderedSet 扩展，使其包含一个对 Swift 类的替代引用，那么也可以使用替代引用来确定存储引用的唯一性。复制一个 OrderedSet 的值将会为它的这两个成员添加新的引用，所以对象的引用计数会保持同步。现在，让我们着手修改 OrderedSet 的定义，为其添加一个额外的成员：

```
private class Canary {}

public struct OrderedSet<Element: Comparable>: SortedSet {
  fileprivate var storage = NSMutableOrderedSet()
  fileprivate var canary = Canary()
  public init() {}
}
```

canary 存在的唯一目的是表示变更 storage 是否安全。(另外一种方法是将 NSMutableOrderedSet 的引用放到新的 Swift 类**内部**。这也可以顺利达到目的。)

现在我们可以定义一个为安全修改存储保驾护航的方法：

```
extension OrderedSet {
  fileprivate mutating func makeUnique() {
    if !isKnownUniquelyReferenced(&canary) {
      storage = storage.mutableCopy() as! NSMutableOrderedSet
```

```
        canary = Canary()
      }
    }
}
```

有一个需要注意的点是，一旦我们发现旧的 canary 过期了，需要立即创建一个新的。假如我们忘了，这个函数会在每次被调用时都复制存储。

至此，实现值语义已经变得手到擒来，只要记住在变更发生之前调用 makeUnique 即可。

插入

最后，让我们来实现 insert。NSMutableOrderedSet 中的 insert 方法很像 NSMutableArray 的，它接受一个整数索引作为参数：

```
class NSOrderedSet: NSObject { // 在 Foundation 中
  ...
  func insert(_ object: Any, at idx: Int)
  ...
}
```

所幸，我们在上面用过的 index(of:inSortedRange:options:usingComparator:) 方法也可以准确地找到我们的新元素应该插入到的索引，以保证不会破坏排序顺序；我们需要做的，就只是将它的 options 参数设置为 .insertionIndex。这样一来，即使元素不在集合中，它也会返回一个有效索引：

```
extension OrderedSet {
  fileprivate func index(for value: Element) -> Int {
    return storage.index(
      of: value,
      inSortedRange: NSRange(0 ..< storage.count),
      options: .insertionIndex,
      usingComparator: OrderedSet.compare)
  }
}
```

准备就绪，我们要开始实现实际的插入了。这并不复杂，只需要将新元素作为参数调用 index(for:)，并检查该元素是否已经存在：

```
extension OrderedSet {
  @discardableResult
  public mutating func insert(_ newElement: Element) -> (inserted: Bool,
  ↪   memberAfterInsert: Element)
  {
    let index = self.index(for: newElement)
    if index < storage.count, storage[index] as! Element == newElement {
      return (false, storage[index] as! Element)
    }
    makeUnique()
    storage.insert(newElement, at: index)
    return (true, newElement)
  }
}
```

我们付出了相当大的努力来实现 makeUnique；如果我们在上面忘记调用它，估计就只能用追悔莫及来形容复杂的心情了。但事实上这个错误很容易发生，然后我们就会疑惑：为什么向一个集合插入值时，偶尔会把其它集合也一同修改了。

结束了！我们现在有了第二个可以愉快玩耍的 SortedSet 实现。

测试

以下是以随机顺序将 1 到 20 之间的数字插入有序集合的代码：

```
var set = OrderedSet<Int>()
for i in (1 ... 20).shuffled() {
  set.insert(i)
}
```

这个集合中的数据应该是有序的；让我们来看看结果是否正确：

```
▶ set
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

太棒了！那 contains 怎么样呢？它能正确地进行查找吗？

```
▶ set.contains(7)
  true
▶ set.contains(42)
  false
```

我们也可以使用 Collection 的方法来操作集合。作为测试，让我们来试一试计算所有元素的总和：

```
▶ set.reduce(0, +)
  210
```

没问题，那我们能得到正确的值语义吗？

```
let copy = set
set.insert(42)
▶ copy
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
▶ set
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 42]
```

看起来也没有问题。会不会感觉很了不起呢？

我们还做了一些额外的工作，来确保我们的 OrderedSet 支持没有实现 Hashable 的 Element，所以现在最好来检查一下是否能正常运作。下面是一个包含单一整型属性的简单可比较结构体：

```
struct Value: Comparable {
  let value: Int
  init(_ value: Int) { self.value = value }
```

```
  static func ==(left: Value, right: Value) -> Bool {
    return left.value == right.value
  }

  static func <(left: Value, right: Value) -> Bool {
    return left.value < right.value
  }
}
```

当我们将 Value 转换为 AnyObject 时，它们会得到一个没有使用 == 的 isEqual 实现，而 hash 属性会返回一些看起来很随机的值，如下所示：

```
let value = Value(42)
let a = value as AnyObject
let b = value as AnyObject
▶ a.isEqual(b)
  false
▶ a.hash
  140261574329264
▶ b.hash
  140261574324016
```

我们可以将这个类型放到前面的例子中试一试，以验证 OrderedSet 并不依赖于哈希：

```
var values = OrderedSet<Value>()
(1 ... 20).shuffled().map(Value.init).forEach { values.insert($0) }
▶ values.contains(Value(7))
  true
▶ values.contains(Value(42))
  false
```

很棒，看起来没有问题。

我认为在本书的 playground 版本中进行测试是个好主意。还可以顺便试着切换到有 bug 的 contains 版本来看一看它会对结果造成怎样的影响。

性能

图 3.2 绘制了 OrderedSet 操作的性能。这个图中最显眼的一个地方是 contains 和 contains2 之间的巨大差距。看来说 Foundation 的 NSOrderedSet.contains 很快并不是在开玩笑：比起二分查找快了大约 15–25 倍。比较悲剧的是，这只针对可哈希的元素...

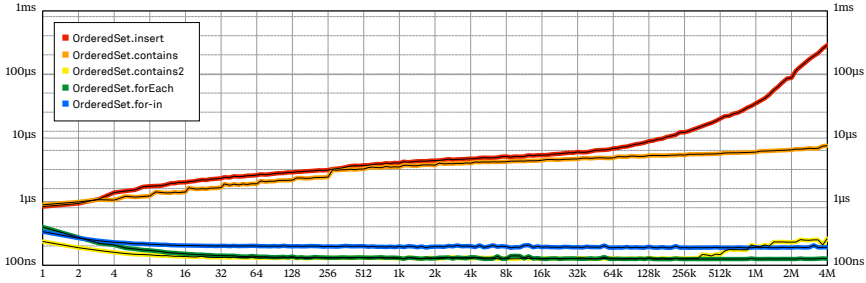


图 3.2: OrderedSet 操作的性能测试结果。该图基于双对数坐标系反映了单次迭代中输入值的元素个数和总体执行时间的关系。

实在有趣，当元素数量超过 16,000 之后，contains2、forEach 和 for-in 似乎全都慢了下来。contains2 在一个哈希表中查找随机值，所以我们好像可以把它的下降原因归结为缓存或页表缓存的颠簸，这和 SortedArray.contains 的情况差不多。但是这个解释在 forEach 和 for-in 上说不过去：它们只是按照元素在集合中出现的顺序进行迭代，按理说曲线应该是完全水平的。如果不对 NSOrderedSet 进行逆向工程，恐怕很难说发生了什么；这简直是一个谜！

OrderedSet.insert 的曲线以二次函数收尾，就像 SortedArray.insert 一样。图 3.3 的两个插入算法的实现相互竞争。很显然，元素数量较少时，NSOrderedSet 的消耗比 Array 大很多，后者较之前者大概快了 64 倍。(部分原因是 NSOrderedSet 需要将元素装箱到一个 NSObject 衍生类型中；将元素类型转换从 Int 转换为一个对整数类型进行简单封装的类，这个类可以将两个算法之间的差距缩小到只有 800%。) 但是在大约 300,000 个元素之后，NSOrderedSet 克服了自身不足，最终反而比 Swift 数组快了 2 倍！

发生了什么？标准库中的 Array 定义有着丰富的语义注释 (semantic annotations)，这有助于编译器以一些几乎想不到的方法来优化代码；编译器还可以内联 Array 的方法，从而消除哪怕仅仅是一个函数调用的微小成本，并探寻进一步优化的可能性。那么，这些弱小且无法优化的 Objective-C 类是怎么做到在插入时反而比 Array 更快的呢？

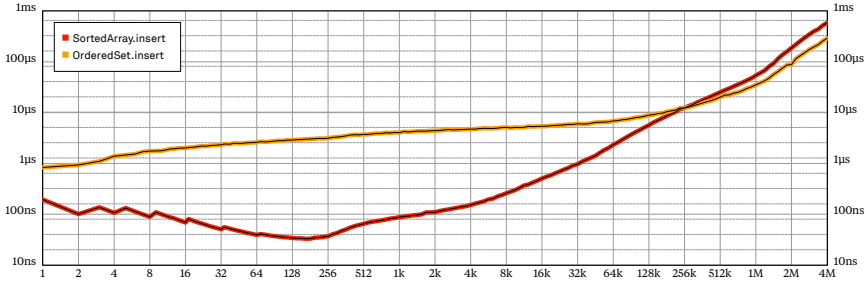


图 3.3: 比较两种 **insert** 实现的性能。

秘密在于，NSMutableOrderedSet 可能是构建在 NSMutableArray 之上的。NSMutableArray 根本就不是一个数组，它是一个双端队列 (double-ended queue)，简称 *deque*。从前面向 NSMutableArray 插入一个新元素与从后面追加一个元素花费的是相同的常量时间。这与 Array.insert 形成了鲜明的对比，其中在位置 0 插入一个元素是 $O(n)$ 操作，因为该操作需要通过将每个元素向右移动一个位置来为新元素腾出空间。

将元素向前或向后移动的过程中，NSMutableArray.insert 需要移动的元素一定少于一半；当我们有足够多的元素时，大部分的插入时间会被移动元素所占据，所以平均看来，尽管 Swift 编译器的优化足够聪明，NSMutableArray.insert 还是会比 Array.insert 快两倍。这太酷了！

总体来说，200% 的提速无法完全弥补 NSOrderedSet 在元素数量较少时缓慢的缺陷。此外，也许并不太引人注目，但插入操作仍然维持着 $O(n^2)$ 的增长率：创建一个包含四百万元素的 OrderedSet 需要花费超过 26 分钟。这可能比 SortedArray 要好 (SortedArray 处理同样的任务大约需要 50 分钟)，但它依旧慢得可怕。

我们已经在 NSOrderedSet 上付出了大量精力，但是得到的回报却不成正比。我们的代码复杂，脆弱，缓慢。不过，这章绝对不能说是**彻头彻尾**的失败，我们又创建了一个正确的 SortedSet 实现，而且，我们学到了一个很好的解决方案，来使用 Swift 封装传统的 Objective-C 接口，这会成为一个长期实用的技能。

有没有办法实现真正意义上比 SortedArray 更快的 SortedSet 呢？答案是肯定的！在此之前，我们需要先学习搜索树。

红黑树

4

自平衡二叉搜索树可以为有序集合类型的实现提供高效的算法。特别是，用这些数据结构来实现的有序集合，其中元素的插入只需消耗对数时间。这实在是一个相当有吸引力的特性，还记得吗，我们实现的 SortedArray 的插入是线性时间复杂度的操作。

“自平衡二叉搜索树” 这样的描述看起来多少有些专业，每个词组都有一个具体的含义，之后我会快速地解释一下它们。

树是一种将数据存储在**节点**内部，按分支排布为树状结构的数据结构。每棵树有一个位于顶部的单独节点，被称作**根节点**。(树的根被置于顶部，追溯历史，计算机科学家们已经将树颠倒着画了几十年了。这并不是因为他们不知道一棵真实的树长什么样，只不过这样更容易画树形图而已。反正，至少我希望是这样的。) 如果一个节点没有子节点，那就将它称为**叶子节点**；否则就是一个**内部节点**。一棵树通常有大量叶子节点。

通常，内部节点可能拥有任意个子节点，但是对于**二叉树**来说，节点只可以拥有**左**和**右**两个子节点。一些节点有两个子节点，当然，只有左子节点或右子节点甚至是根本没有子节点的情况也是时常存在的。

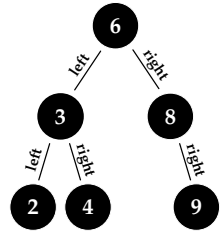


图 **4.1:** 一棵二分搜索树。节点 **6** 是根节点，节点 **6**、**3** 和 **8** 是内部节点，而节点 **2**、**4** 和 **9** 是叶子。

在**搜索树**中，节点内部的值在某种程度上是可比较的，而且树中的节点都是按照一定次序排列的，所有左子树中的值都比节点自身的小，右子树则相反，比节点自身的值更大。这使得查找任意指定元素变得很容易。

通过**自平衡** (意味着这个数据结构有排序机制)，无论一棵树包含什么值，以及这些值以什么顺序被插入，这棵树的高度都可以确保尽可能低，且在此范围内保持完整而茂密。如果允许树肆意地生长，那么很简单的操作都可能变得效率奇低。(举一个极端的例子来说，如果一棵树所有的节点最多都只有一个子节点，如同链表一般，那可以说是根本没有效率。)

创建自平衡二叉树的方法有很多；在这个部分中，我们将会实现一个名叫**红黑树**的版本。由于红黑树自身独有的特征，为了实现自平衡的部分，每个字节都需要额外多存储一位来保存相关信息。这额外的一位是节点的颜色，可以是红色或黑色。

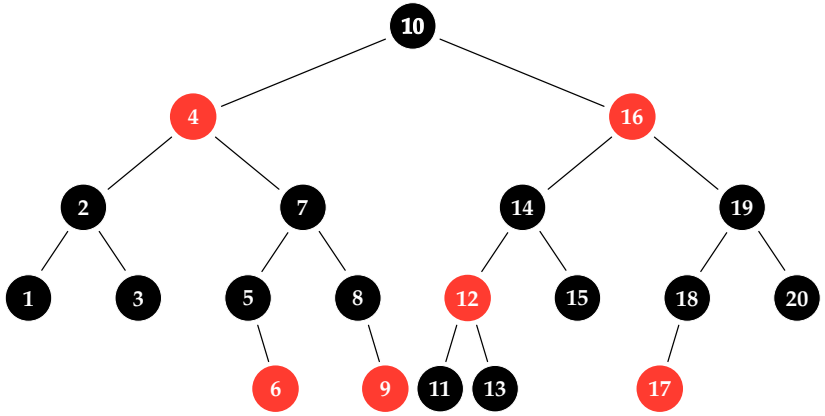


图 4.2: 一棵示例红黑树。

红黑树总是保持它的节点的按照一定顺序排布，并以恰当的颜色着色，从而始终满足下述几条性质：

1. 根节点是黑色的。
2. 红色节点只拥有黑色的子节点。(只要有，就一定是。)
3. 从根节点到一个空位，树中存在的每一条路径都包含相同数量的黑色节点。

空位指的是在树中所有可以插入新节点的空间，即，一个左右子节点都没有的节点。要让增长一个节点，我们只需要用一个新节点替换它的一个空位即可。

第一个性质使得算法略微简单了一点；而且完全不会影响树的形态。后两个性质保证了树的密度始终良好，树中的空位与根节点的距离，不会超过其它任意节点与根节点距离的两倍。

为了完全理解这些平衡性质，稍微做几个小实验，探索一下它们的极端情况，可能会很有帮助。例如，可以构建一棵只包含黑色节点的红黑树；图 4.3 中的树就是一个例子。

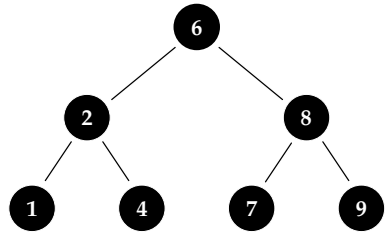


图 4.3: 每个节点都是黑色的示例红黑树。

如果我们尝试构建更多的例子，很快就会意识到红黑树的第三条性质其实将这种树限定为了一种特定的形态：所有内部节点都有两个子节点，而且所有叶子节点都在同一层上。形态如此的树被称为**完美树**，因为它们完全平衡，完全对称。我们期望所有平衡树都这样生长成这样的理想形态，因为它的每个节点都已尽可能靠近根节点。

不过，要求平衡算法来维护完美树是不可能的：实际上，只有特定的节点数才能构建完美搜索树。比方说，没有哪棵完美树拥有四个节点。

为了使红黑树更实用，第三条性质使用了一个平衡的弱定义，那就是红色节点并不会被计算。不过，为了不让事情变的**难以控制**，第二条性质将红色节点的数量限制在了合理范围内：这确保了在树中的任何指定路径上，红色节点的数量都不会超过黑色节点。

代数数据类型

现在关于红黑树应该是什么，我们有了一个粗略的想法，让我们从一个节点的颜色开始，马上进入实现环节吧。传统上通常会使用低层次 hack 的方式将颜色信息放到节点的二进制表示的一个未使用的位中，这样的话可以不占用额外的空间。但是我们喜欢干净且安全的代码，所以使用枚举来表示颜色才是我们的心之所向：

```
public enum Color {
  case black
  case red
}
```

> Swift 编译器有时自己就能够将颜色的值放到这样一个未使用的位，而这并不需要我们做任何事。相较于 C/C++/Objective-C，Swift 类型的二进制布局非常灵活，而且编译器拥有很大程度的自由来决定如何进行打包。对于具有关联值的枚举来说尤其如此，编译器经常能够找到未使用的位模式来表示枚举成员，而无需分配额外的空间来区分它们。例如，Optional 会将一个引用类型封装到其自身的空间中。Optional.none 成员则由一个从来没有被任何有效引用使用过的 (全零) 位模式来表示。(顺便一提，相同的全零位模式也用来表示 C 的空指针和 Objective-C 的 nil 值，这在某种程度上提供了二进制兼容性。)

一棵树本身要么是空的，要么含有一个根节点，这个根节点具有颜色，且包含一个值和左右两个子节点。Swift 允许枚举的成员包含字段值，这能够将该描述转换为第二个枚举类型：

```
public enum RedBlackTree<Element: Comparable> {
  case empty
  indirect case node(Color, Element, RedBlackTree, RedBlackTree)
}
```

在实际的代码中，我们常常会给一个节点的字段加上标签，这样它们所扮演的角色就很清晰了。这里我们没有给它们命名，单纯只是为了避免下面的示例代码在不合适的地方被换行。对此我深感抱歉。

我们之所以需要使用 indirect case 语法，是因为节点的子节点是树本身。indirect 关键字强调了在我们的代码中递归的存在，而且也允许编译器将节点的值装箱到隐藏的在堆上申请内存的引用类型中。(这么做是必须的，它可以防止不必要的麻烦，比如编译器无法将特定的存储大小分配给枚举值。递归这种数据结构有时候很是棘手。)

拥有像这样的字段的枚举体现了 Swift 定义**代数数据结构**的方式，(马上就会看到) 这为我们提供了一种强大而优雅的构建数据结构的方式。

至此新红黑树类型的骨架搭建已经完成，我们准备进一步在树上实现 SortedSet 的方法。

模式匹配和递归

当使用代数数据类型的时候，我们一般使用模式匹配来将我们想要的解决的问题拆分为许多不同的情况，然后一一解决各种不同的情况。对于某一种特定情况来说，通常我们会依赖递归的方式来解决这种情况中稍小的版本。

contains

举个例子，让我们来看一看 contains 的实现。我们的搜索树按照特定的顺序存储值，据此我们可以将查找一个元素这个问题分割为四种小情况：

1. 如果树为空，则它不包含任何元素，contains 一定返回 false。
2. 或者，树的顶端肯定有一个根节点。如果存储的元素恰好等于我们正在查找的元素，那么我们就知道了这棵树包含元素；这种情况下 contains 应该返回 true。
3. 或者，如果根值大于我们正在查找的元素，那么该元素一定不存在于右子树中。当且仅当元素存在于左子树，树才包含该元素。
4. 或者，根值比元素小，那么在右子树中进行查找即可。

我们可以使用 switch 语句将上面的文字描述直接转换为 Swift 表达式：

```
public extension RedBlackTree {
  func contains(_ element: Element) -> Bool {
    switch self {
    case .empty:
      return false
    case .node(_, element, _, _):
      return true
    case let .node(_, value, left, _) where value > element:
      return left.contains(element)
    case let .node(_, _, _, right):
      return right.contains(element)
    }
  }
}
```

用 Swift 的模式匹配语法表达这类结构条件非常自然。我们将在 RedBlackTree 上定义的大多数方法都会遵循这种结构，当然了，细节上各有不同。

forEach

在 forEach 中，我们想按照升序在树中对所有元素调用一个闭包。如果树为空，没有任何难度就可以做到：因为根本就什么都不用做。除此之外，我们需要先访问所有左子树中的元素，然后是存储在根节点中的元素，最后访问右子树中的元素。

这样的场景很适合使用 switch 语句的另一种递归方法：

```
public extension RedBlackTree {
  func forEach(_ body: (Element) throws -> Void) rethrows {
    switch self {
    case .empty:
      break
    case let .node(_, value, left, right):
      try left.forEach(body)
      try body(value)
      try right.forEach(body)
    }
  }
}
```

这个例子甚至比 contains 还要简短，但是你会发现它们有着相同的构造，类似的风格：在 switch 语句中有几个模式匹配的分支，以及在分支内部偶尔使用递归。

这个算法对树进行了**中序遍历** (inorder walk)。记住这个词会很有用；如果你打算去参加面试，而考官出题很可能就涉及到这样的名词，要是你不知所云的话，那可就完蛋了。当我们在一棵搜索树中进行中序遍历时，我们将会从最小值到最大值 “按照顺序” 访问元素。

树形图

接下来，我们将会稍微绕一点儿路，用一个很有意思的自定义方法来实现 CustomStringConvertible。(我希望你还记得，我们曾经在引言中写过一个泛型的版本。)

开发一个新的数据结构时，投资一些额外的时间来保证我们可以用一个简单易懂的格式来输出值的确切结构是非常有价值的。这份投资通常会让调试变得容易很多，可以让你事半功倍。至于红黑树，我们会**尽可能**努力，使用 Unicode 图形来构建精细的小规模树形图。

下面的属性返回代表 Color 的符号，我们以此作为开始。目前黑白小方块表现尚可：

```
extension Color {
  var symbol: String {
    switch self {
    case .black: return "■"
    case .red:   return "□"
    }
  }
}
```

我们将会巧妙地修改 forEach 函数来让它自己生成图。为了让本书不那么无聊，我决定把读懂这段代码的机会留给你，也许这能让你更容易弄清楚到底发生了什么。我保证，它绝不像看起来那样难以下手！(提示：可以尝试改变一些字符串文字，看看下面的输出结果会有什么变化。)

```
extension RedBlackTree: CustomStringConvertible {
  func diagram(_ top: String, _ root: String, _ bottom: String) -> String {
    switch self {
    case .empty:
      return root + "•\n"
    case let .node(color, value, .empty, .empty):
      return root + "\(color.symbol) \(value)\n"
    case let .node(color, value, left, right):
      return right.diagram(top + "    ", top + "┌───", top + "│   ")
        + root + "\(color.symbol) \(value)\n"
        + left.diagram(bottom + "│   ", bottom + "└───", bottom + "    ")
    }
  }

  public var description: String {
    return self.diagram("", "", "")
  }
}
```

让我们来看看对于一些简单的树，diagram 做了什么。RedBlackTree 只是一个枚举，所以我们可以通过手动嵌套枚举来构建各种各样的树。

1. 一棵空树被打印为一个小黑点。它是由上述 diagram 方法的第一个 case 模式实现的：

```
let emptyTree: RedBlackTree<Int> = .empty
▶ emptyTree
  •
```

2. 只有一个节点的树会匹配第二个 case 模式，所以它被打印为一条由节点颜色和值组成的线上：

```
let tinyTree: RedBlackTree<Int> = .node(.black, 42, .empty, .empty)
▶ tinyTree
  ■ 42
```

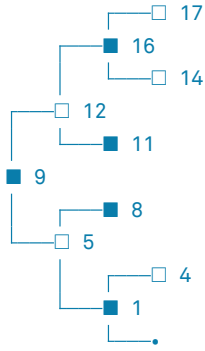
3. 最后，一棵拥有根节点且包含非空子节点的树与第三个 case 相匹配。它的描述看起来和前一个 case 很相似，不过根节点已经生长出表示其子节点的左右枝：

```
let smallTree: RedBlackTree<Int> =
  .node(.black, 2,
    .node(.red, 1, .empty, .empty),
    .node(.red, 3, .empty, .empty))
▶ smallTree
  ┌──□ 3
  ■ 2
  └──□ 1
```

即使表示更复杂的树，也一样没问题，图形也确实像一棵树：

```
let bigTree: RedBlackTree<Int> =
  .node(.black, 9,
    .node(.red, 5,
      .node(.black, 1, .empty, .node(.red, 4, .empty, .empty)),
      .node(.black, 8, .empty, .empty)),
    .node(.red, 12,
      .node(.black, 11, .empty, .empty),
      .node(.black, 16,
        .node(.red, 14, .empty, .empty),
        .node(.red, 17, .empty, .empty))))
```

▶ bigTree



是不是还挺齐整的？有时使用代数数据结构就是这么神奇，本该很复杂的事情，在你不经意之间就完成了。

现在我们要回到编码中去了，让我们继续来构建一个有序集合的实现。

插入

在 SortedSet 中，我们将插入定义为了一个可变函数。不过，对于红黑树的例子来说，我们将会定义一个更简单的函数式版本，该版本不会对已经存在的树进行修改，它将返回一棵全新的树。下面是为它量身打造的函数签名：

```
func inserting(_ element: Element) -> (tree: RedBlackTree, existingMember: Element?)
```

拥有这样的一个函数，我们就可以通过将该函数返回的树赋值给 self 自身来实现可变插入了：

```
extension RedBlackTree {
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪  Element)
  {
    let (tree, old) = inserting(element)
    self = tree
```

```
    return (old == nil, old ?? element)
  }
}
```

对于 inserting，我们将参照由 Chris Okasaki 在 1999 年 首次发表的一个非常出色的模式匹配算法来进行实现。

务必牢记在心，红黑树需要满足下述三个要求：

1. 根节点是黑色的。
2. 红色节点只拥有黑色的子节点。(只要有，就一定是。)
3. 从根节点到任意一个空位，树中存在的每一条路径都包含相同数量的黑色节点。

保证满足第一个要求的方法是，当作要求并不存在，直接插入元素。如果得到的树的根节点恰好是红色，则将它染黑即可，这么做并不会影响其它要求。(第二个要求只关心红色节点，所以我们可以将每一个节点都绘制为黑色，这样并不会破坏这条要求。对于第三个要求：根节点存在于树中的每一条路径上，所以将它染黑能够统一增加所有路径上包含的黑色节点数量。因此，如果树在为根节点染色之前本来就已经满足第三个要求的话，那么重新将树进行染色也并不会违反这个条件。)

所有的这些考虑指引着我们将 inserting 定义为一个致力于保证第一个要求的短小的封装函数。它将实际的插入操作委托给一些我们尚未定义的内部辅助函数：

```
extension RedBlackTree {
  public func inserting(_ element: Element) -> (tree: RedBlackTree, existingMember:
  ↪  Element?) {
    let (tree, old) = _inserting(element)
    switch tree {
    case let .node(.red, value, left, right):
      return (.node(.black, value, left, right), old)
    default:
      return (tree, old)
    }
  }
}
```

接下来让我们处理一下 _insertion 方法。它的基本工作是查找指定元素可以作为叶子节点被插入到树中的位置。这个任务和 contains 很像，这是由于我们需要沿着与查找已存在的元素时相同的路径一路向下。所以发现 _insertion 和 contains 有着完全相同的结构也并不是什么震惊的事情；它们只需要在各个 case 语句返回**略微**不同的东西就行了。

让我们依次来看一看四个例子，与此同时，我们将对代码进行解释：

```
extension RedBlackTree {
  func _inserting(_ element: Element) -> (tree: RedBlackTree, old: Element?)
  {
    switch self {
```

首先，向一棵空树插入一个新的元素，我们只需简单地创建一个包含指定值的根节点：

```
    case .empty:
      return (.node(.red, element, .empty, .empty), nil)
```

显而易见，上述代码违反了第一个要求，因为我们从一个红色节点开始创建树。但是这算不上问题，当我们返回之后 inserting 将会对其进行修正。另外，代码满足其它两个要求。

让我们继续看第二个例子。如果我们试图插入的值与根节点所持有的相同，则说明树已经包含该值，于是我们可以安全地返回 self 和现有成员的复制：

```
    case let .node(_, value, _, _) where value == element:
      return (self, value)
```

由于 self 理应满足所有条件，不修改直接返回并不会破坏任何条件。

除此之外，取决于与根节点值比较的结果，值最终应该被插入左子树或右子树，所以我们需要做一个递归调用。如果返回值表明值尚未存在于树中，那么我们需要返回一个根节点的复制，相同子树的前一个版本会被这个新的替代。(否则，我们只需再次返回 self。)

```
    case let .node(color, value, left, right) where value > element:
      let (l, old) = left._inserting(element)
      if let old = old { return (self, old) }
```

```
        return (balanced(color, value, l, right), nil)

      case let .node(color, value, left, right):
        let (r, old) = right._inserting(element)
        if let old = old { return (self, old) }
        return (balanced(color, value, left, r), nil)
      }
    }
}
```

你也许已经注意到了，上面的两个例子都在返回新树之前调用了一个神秘的 balanced 函数。这个函数暗藏着红黑树的魔法。(这里的 _inserting 函数与我们可能会为普通二叉搜索树定义的插入操作几乎一模一样；事实上，在这里红黑树特定的东西并不多。)

平衡

balanced 方法的工作是检测现有的树是否违反了平衡的要求，如果是，则巧妙地重排节点，随即返回符合标准的树来进行修复。

上面的 _inserting 代码创建了一个新的节点作为红色叶子节点；在大多数例子中，这个处理会在一个递归调用中完成，然后将结果作为一个已经存在的节点的子节点插入。这样做哪些红黑树的要求可能会被违反呢?

第一个要求很安全，因为它只和根节点相关，不会影响子节点 (另外，我们在 inserting 中特别关照了根节点，不管怎么说这里我们都可以放心地忽略它)。由于第三个性质并不关心红色节点，插入一个新的红色叶子节点也不会破坏它。然而，代码中并没有任何举措能防止我们在一个红色亲节点下插入同为红色的子节点，所以最终第二个性质可能会被违反。

所以 balanced 只需要检查第二项要求，然后在不破坏第三项的同时设法修复它。在一棵合法的红黑树中，一个红色节点总会拥有一个黑色的亲节点；如果插入操作违反了第二个要求，那么其结果一定会匹配 图 4.4 中的某一个例子。

完成插入之后，我们可以通过检查树是否匹配这些模式，来实现重新平衡的处理，如果匹配，重新组织树，使其匹配模式 R。由此产生的模式在巧妙修复了第二个要求的同时，又不会破坏第三个要求。

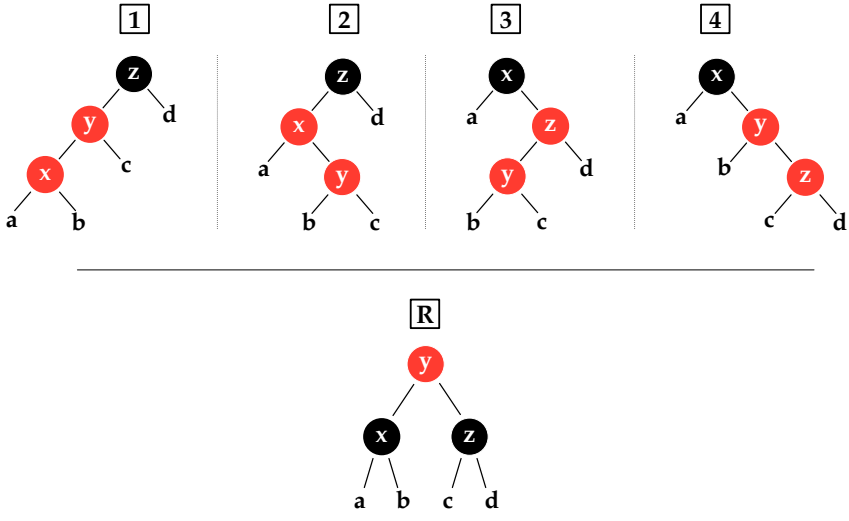


图 4.4: 插入一个元素之后，一个红色节点拥有红色子节点的四种可能的情况。*x*、*y* 和 *z* 代表值，而 *a*、*b*、*c* 和 *d* 是 (可能为空的) 子树。如果树与 1-4 的模式相匹配，则它的节点需要按照模式 R 重组。

你能猜到我们将会使用什么工具来进行实现吗?

由于某种惊人的巧合, 这个特殊的问题可以称得上是极好地展示了在代数数据类型上模式匹配的威力。作为开始，让我们将 图 4.4 中的五个图转换为 Swift 表达式。当我们创建华丽的 Unicode 树时，我们看到了如何精心使用 Swift 表达式来构建小规模的树；现在我们只需要将这些知识运用到下面的图中即可：

```
1: .node(.black, z, .node(.red, y, .node(.red, x, a, b), c), d)
2: .node(.black, z, .node(.red, x, a, .node(.red, y, b, c)), d)
3: .node(.black, x, a, .node(.red, z, .node(.red, y, b, c), d))
4: .node(.black, x, a, .node(.red, y, b, .node(.red, z, c, d)))
R: .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
```

此刻，我们基本完成了！只需要将这些表达式添加到一个 switch 语句中，我们就能得到一个正确的 balanced 函数实现：

```
extension RedBlackTree {
```

```
  func balanced(_ color: Color, _ value: Element, _ left: RedBlackTree, _ right:
  ↪  RedBlackTree) -> RedBlackTree {
    switch (color, value, left, right) {
    case let (.black, z, .node(.red, y, .node(.red, x, a, b), c), d),
      let (.black, z, .node(.red, x, a, .node(.red, y, b, c)), d),
      let (.black, x, a, .node(.red, z, .node(.red, y, b, c), d)),
      let (.black, x, a, .node(.red, y, b, .node(.red, z, c, d))):
      return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
    default:
      return .node(color, value, left,  right)
    }
  }
}
```

这是一种十分出色的编程方法。本质上，我们用这样一种方法将图 4.4 直接翻译为了 Swift，这样一来，我们用来解释任务的五个图在代码里仍然能被识别出来！

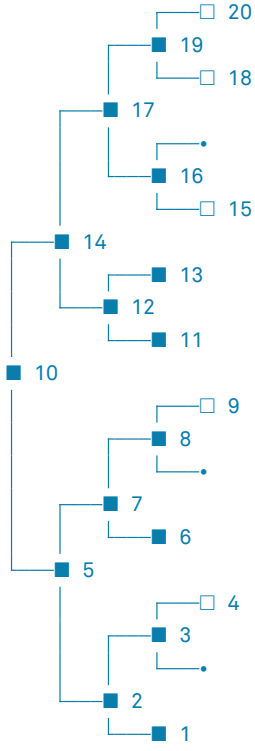
有一个小问题。虽然上述 Swift 代码完全有效，但很遗憾，它在 Swift 4 的编译器中会发生崩溃。为了解决这个错误，我们需要将所有情况分开处理，只能重复写四个几乎一样的 case 语句：

```
extension RedBlackTree {
  func balanced(_ color: Color, _ value: Element, _ left: RedBlackTree, _ right:
  ↪  RedBlackTree) -> RedBlackTree {
    switch (color, value, left, right) {
    case let (.black, z, .node(.red, y, .node(.red, x, a, b), c), d):
      return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
    case let (.black, z, .node(.red, x, a, .node(.red, y, b, c)), d):
      return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
    case let (.black, x, a, .node(.red, z, .node(.red, y, b, c), d)):
      return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
    case let (.black, x, a, .node(.red, y, b, .node(.red, z, c, d))):
      return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
    default:
      return .node(color, value, left, right)
    }
  }
}
```

这个变通的方法淡化了一丝原来的震惊，幸亏魔法依然闪耀。(衷心希望这个 bug 在未来的编译器中得以被修复。)

来看一看插入操作是否真的有效，创建一棵包含数字 1 到 20 的红黑树：

```
var set = RedBlackTree<Int>.empty
for i in (1 ... 20).shuffled() {
  set.insert(i)
}
▶ set
```



看起来一切正常！欢呼起来！可以看到我们的树满足所有三个红黑树应有的性质。

在插入数字之前，我们先调用了 shuffled 函数，所以每次执行这段代码都能够得到全新的树。借助本书的 playground 版本来进行测试会是一个绝佳的选择。

集合类型

实现像 Collection 一样的协议恐怕是代数数据类型开始显得不那么方便的地方。我们需要定义一个合适的索引类型，而最简单的方法就是直接使用一个元素本身作为索引，就像这样：

```
extension RedBlackTree {
  public struct Index {
    fileprivate var value: Element?
  }
}
```

value 是一个可选值，因为我们将会使用 nil 值来表示结束索引。

集合类型的索引必须是可以比较的。幸运的是，我们的元素正好是可比较的，这样比较索引就相当容易了。唯一棘手的事情是，我们必须确保结束索引比其它任何索引都**更大**：

```
extension RedBlackTree.Index: Comparable {
  public static func ==(left: RedBlackTree<Element>.Index, right:
  ↪  RedBlackTree<Element>.Index) -> Bool {
    return left.value == right.value
  }

  public static func <(left: RedBlackTree<Element>.Index, right:
  ↪  RedBlackTree<Element>.Index) -> Bool {
    if let lv = left.value, let rv = right.value {
      return lv < rv
    }
    return left.value != nil
  }
}
```

接下来，我们要具体实现 Sequence 扩展方法中的 min() 和 max()，以取得树的最小元素和最大元素。它们是在下面即将实现的索引步进的必要组成部件。

最小的元素是存储在最左边的节点中的值；这里我们使用模式匹配和递归来查找它：

```
extension RedBlackTree {
  func min() -> Element? {
    switch self {
    case .empty:
      return nil
    case let .node(_, value, left, _):
      return left.min() ?? value
    }
  }
}
```

查找最大元素可以用相似的方法。不过为了让事情更有趣一些，这里的 max() 版本会把递归展开成为一个循环：

```
extension RedBlackTree {
  func max() -> Element? {
    var node = self
    var maximum: Element? = nil
    while case let .node(_, value, _, right) = node {
      maximum = value
      node = right
    }
    return maximum
  }
}
```

注意，理解这段代码比 min() 更难。这个版本并没有使用一个简单的表达式来定义结果，而是使用了一个 while 循环，其中一些内部状态一直在改变。为了理解它是如何工作的，你需要在大脑中运行代码，来弄清楚 node 和 maximum 是如何随着循环的进程而改变的。但是从根本上说，这不过是相同算法的两种不同表达方式罢了，两者都有存在的意义。迭代版本有时稍微要快一点，所以可以通过使用它，用一点可读性交换一点额外的性能。不过递归的解决方案往往会更容易上手。如果性能测试告诉我们降低可读性是值得的，我们随时可以重写代码。

现在我们拥有了 min() 和 max() 方法，可以开始实现 Collection 了！

首先从最基础的开始：startIndex、endIndex 和 subscript：

```
extension RedBlackTree: Collection {
  public var startIndex: Index { return Index(value: self.min()) }
  public var endIndex: Index { return Index(value: nil) }

  public subscript(i: Index) -> Element {
    return i.value!
  }
}
```

Collection 的实现应该明确定义关于索引失效的规则。在我们的例子中，我们可以对索引的有效性进行定义：对于一个原本是在树 t 中所创建的索引 i，如果有另一棵树 u，那么当且仅当 i 是结束索引，或者 t[i] 的值存在于 u 中时，i 是 u 中的**有效索引**。这个定义允许人们在树发生某些改变之后重用一部分索引，这是很有用的特性。(该规则与 Array 索引的工作方式不同，因为在我们的例子中，索引基于值，而非位置。这种定义索引无效的方式有点不同于寻常，但是并不违反 Collection 的语义要求。)

我们的下标实现特别简短，因为只是解包索引中的值而已。然而，这么做不太好，它并没有验证存储在索引中的值是否确确实实存在于树中。我们可以用任意索引对任意的树进行下标操作，然后得到一个可能有用也可能并不存在的结果。虽然在实现 Collection 的时候我们可以自己定义索引无效的相关规则，但是在运行的时候应该尽可能验证遵循这些规则是否被真正遵守。使用一个无效索引来对集合类型做下标操作是个严重的编码错误，最好通过异常机制来进行处理，而不要默默地返回一些诡异的值。现在就需要这么做。我保证从这里开始我们会做得更好。

不太妙的是，startIndex 调用了 min()，这是一个对数操作。但是 Collection 要求 startIndex 和 endIndex 的实现应该在常数时间内完成，对数操作自然就成为了一个问题。一个相对简单的解决方案是，通过引入一个简单的结构体对树进行封装，来缓存树的节点内部或附近的最小值。(亲爱的读者们，我决定把这个实现留给你们作为一个练习。当然我们也可以假装这一段所说的事情从未发生，继续前进！)

不管怎么说，我们现在需要实现 index(after:)，也就是说，我们要查找树中存在的比指定值大的最小的元素。为此我打算写一个工具函数来达成目的。为了保证我们关于索引验证的承诺，这个函数还会返回一个布尔值，表明存储在索引中的元素是否也存在于树中，这样我们就可以为它设置一个 precondition 来抛出异常：

```
extension RedBlackTree: BidirectionalCollection {
  public func index(after i: Index) -> Index {
    let v = self.value(following: i.value!)
    precondition(v.found)
```

```
      return Index(value: v.next)
    }
  }
```

value(following:) 函数是一个 contains 的巧妙变体。虽然它的逻辑非常复杂，但是绝对值得你再看一次：

```
extension RedBlackTree {
  func value(following element: Element) -> (found: Bool, next: Element?) {
    switch self {
      case .empty:
        return (false, nil)
      case .node(_, element, _, let right):
        return (true, right.min())
      case let .node(_, value, left, _) where value > element:
        let v = left.value(following: element)
        return (v.found, v.next ?? value)
      case let .node(_, _, _, right):
        return right.value(following: element)
    }
  }
}
```

最复杂的部分应该是，第三个分支中，当元素存在于左子树时，会发生什么。通常跟随 (following) 该元素的元素也在同一棵子树中，所以一个递归调用会返回正确的结果。但是若 element 是 left 中的最大值，这次调用会返回 (true, nil)。在这个例子中，我们需要返回跟随 left 子树的值，也就是存储在亲节点中的它自身的值。

除此以外，我们也需要能够用其它方法查找上述的值。同样地，为了让事情有趣，我们将会再次尝试用迭代而非递归来实现一个新版本的方法：

```
extension RedBlackTree {
  func value(preceding element: Element) -> (found: Bool, next: Element?) {
    var node = self
    var previous: Element? = nil
    while case let .node(_, value, left, right) = node {
```

```
        if value > element {
          node = left
        }
        else if value < element {
          previous = value
          node = right
        }
        else {
          return (true, left.max() ?? previous)
        }
      }
      return (false, previous)
    }
}
```

注意这里的 previous 变量的目的是什么，事实上它等效于上面那个复杂例子中的 nil 聚合 (nil-coalescing) 运算符 ??。(我们无法将这种递归结果进行后置处理的代码直接转换为迭代代码，但是可以通过添加新的变量来消除后置处理，这样一来，就可以在从递归调用返回之前进行处理。上面迭代代码中的 previous 就是一个这样的变量。)

距离完成 BidirectionalCollection 的实现，就只差添加调用 value(preceding:) 方法来查找索引的前序了：

```
extension RedBlackTree {
  public func index(before i: Index) -> Index {
    if let value = i.value {
      let v = self.value(preceding: value)
      precondition(v.found)
      return Index(value: v.next)
    }
    else {
      return Index(value: max()!)
    }
  }
}
```

像这样定义 index(after:) 和 index(before:) 比起我们前面所做的明显更加复杂。最终的运行时结果可能也会相当慢 – 毕竟所有递归查找导致算法开销达到了 $O(\log n)$，而非一贯的 $O(1)$。这没有违反任何 Collection 的规则；只不过使用索引进行迭代操作会比一般的集合更慢。

最后，让我们来看看 count 的实现；这是目前为止又一个极好的实践模式匹配和递归的机会：

```
extension RedBlackTree {
  public var count: Int {
    switch self {
    case .empty:
      return 0
    case let .node(_, _, left, right):
      return left.count + 1 + right.count
    }
  }
}
```

如果我们忘记专门实现 count，它的默认实现将会计算 startIndex 和 endIndex 之间的步数，这样 $O(n \log n)$ 比起我们的 $O(n)$ 会慢得多。但是我们的实现也并没有什么值得宣扬的：它仍然需要访问树中每一个节点。

> 我们可以让所有节点记住在它们下面的元素数量，这将使我们的树变成一棵所谓的**顺序统计树** (order statistic tree)，速度也会随之得到提升。(我们将会需要更新插入和任何其它可变方法，并谨慎地维护这个额外的信息。使用这类扩展叫做**树的扩充**。添加元素数量只是众多扩充树的方法中的一种。) 除了提升 count 的速度以外，顺序统计树还带来了许多有趣的附加收益；例如，在这样一棵树中，只需 $O(\log n)$ 个步骤即可查找第 i 小或第 i 大的值。不过，这可不是免费得来的：内存消耗的增加和代码更加复杂便是代价。

大功告成了，在我们最熟知和喜爱的标准库中的扩展实现的帮助下，RedBlackTree 现在已经是一个 BidirectionalCollection 了：

```
▶ set.lazy.filter { $0 % 2 == 0 }.map { "\($0)" }.joined(separator: ", ")
  2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

为了做到有始有终，我们将会实现 SortedSet。幸运的是，除了空树的初始化方法外，我们已经满足了所有的要求 – 不过添加这个方法也并不是什么大工程：

```
extension RedBlackTree: SortedSet {
  public init() {
    self = .empty
  }
}
```

全部完成！为了完善这一章，你可以进行一个挑战：尝试实现 SetAlgebra 的 remove(_:) 操作。专业建议：在第一个版本中姑且先忽略红黑树的要求。在代码能够工作之后，再开始思考如何在删除操作后重新平衡树。(你可以在网上或是喜爱的算法书籍中阅读相关内容，即便是这样这个游戏也不失公平；就算是参考资料，实现这些东西也足够具有挑战性。)

性能

我们预计 RedBlackTree 上的大多数操作都会消耗 $O(\log n)$ 的时间, 除了 forEach, 它应该是 $O(n)$ – 因为它对树中的每个节点 (和空位) 都会执行一次递归调用。运行我们的常规性能测试证明了推测大抵是正确的；图 4.5 绘制了在我的 Mac 上产生的结果。

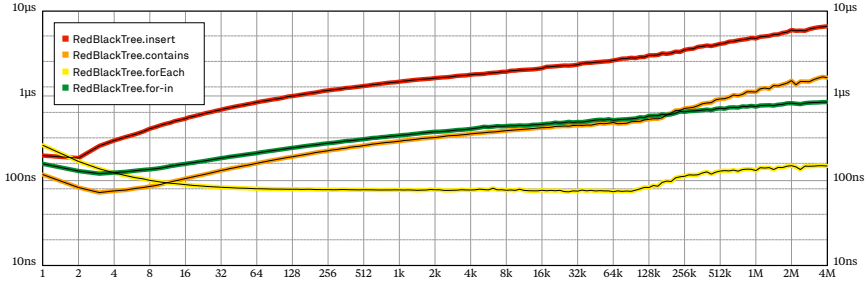


图 4.5: RedBlackTree 操作的性能测试结果。图表在双对数坐标系上展示了输入值的元素个数和单次迭代的平均执行时间。

曲线的噪声很严重。这是因为红黑树允许树的形态发生巨大改变。树的深度不仅取决于元素的个数，我们插入元素的 (随机) 顺序 (在很小的程度上) 也会产生影响。大多数操作的性能与树的深度成正比，所以我们的曲线发生了抖动。不过小规模的噪声并不影响总体形状。

使用 forEach 进行迭代是一个复杂度为 $O(n)$ 的操作，所以所以它的平摊图应该是一条水平线。它的中间部分的确看起来是平坦的，但是大约从 20,000 个元素开始，它渐渐地减速，并开始以对数速率增长。这是因为即使 forEach 以升序访问元素，它的在内存中的位置从本质上来说仍然是随机的：因为在插入初期元素被分配时这一切就已经被决定的。红黑树在保持相邻元素靠近彼此这件事情上，可以说是表现十分糟糕，受困于此，使得它们相当不适合今天的计算机内存架构。

但是不管怎么说，forEach 仍然比 for-in 略快一点；增加索引是一个相对复杂的 $O(\log n)$ 操作，尽管中序遍历总是清楚地知道在哪里找到下一个值，但基本上这个操作都需要从头开始查找一个元素。

contains 与 for-in 之间越来越大的差距是缓存效果的另一个表现。在 for-in 中我们连续调用 index(after:) 来访问随机内存中的新元素，但是大部分路径保持不变，所以 for-in 仍然相对是缓存友好的。contains 就不是这样了，它每次都会迭代访问树中全新的随机路径，内存访问几乎没有重复，从而导致缓存无效。

为了看一看 RedBlackTree 中基于索引的迭代到底多慢，将它与以前的实现进行比较不失为一个好主意。结果如 图 4.6，一个对 RedBlackTree 所有元素进行 for-in 循环的操作比对 SortedArray 的相应操作慢了大约 200–1,000 倍，而且曲线慢慢分离，证明了我们预计的 $O(\log n)$ 和 $O(1)$ 的时间复杂度。

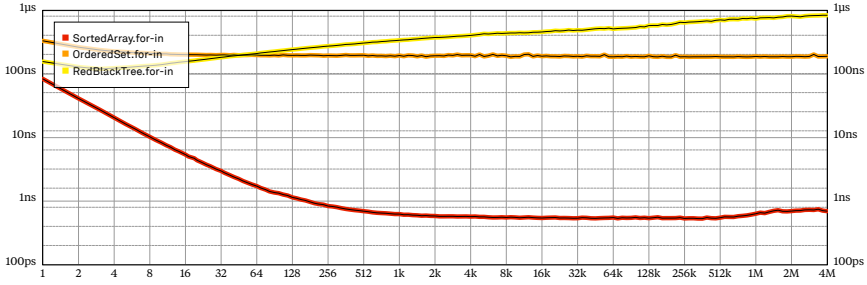


图 4.6: 比较三个 **index(after:)** 实现的性能。

我们会在未来的章节中修复缓慢的问题，但是至少有一部分 (看起来) 应该是由于查找和可变性之间无法避免的权衡所导致的：当元素在巨大的连续缓冲区按顺序排列时，迭代的速度是最快的，但是恰恰这种缓冲区在做快速插入时是最不便利的数据结构。任何对插入性能进行的优化可能都涉及将存储分解成更小的块，而副作用正是迭代速度下降。

在红黑树中，元素被单独封装在树的节点中，这些节点在堆上申请的内存是分离的，如果你想快速地迭代所有元素，这大概是最糟糕的元素存储方式了。这种特殊的设计中，我们使用 $O(\log n)$ 的算法来查找下一个索引，但是也许会让你惊讶，这个算法并不是 for 循环中花费大多数时间的地方：即使用 forEach 来替代 for-in，性能测试中 RedBlackTree 仍然比 SortedArray 慢 90–200 倍。

唉，真令人失望。但是插入变快了，对吧？让我们来看一看！图 4.7 将 RedBlackTree.insert 的性能与之前的实现进行了比较。

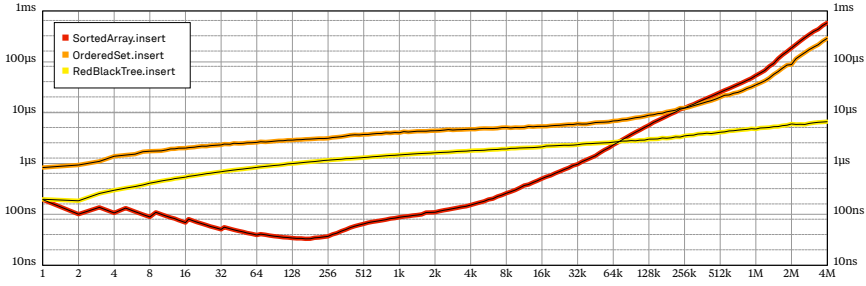


图 **4.7:** 比较三个 **insert** 实现的性能。

我们可以看到，当元素个数变得庞大时，红黑树算法的优势明显得到了回报。向一棵红黑树插入四百万个元素大约比有序数组要快 50–100 倍。(它仅仅花费了 33 **秒**，比 OrderedSet 的 26 **分钟**好太多了。) 如果添加更多元素，两者之间的差距会越来越大。

然而，当我们的元素数量低于 50,000 个时，SortedArray 显然击败了 RedBlackTree。相较于红黑树，有序数组有着更好的位置性：即使我们需要进行更多次的内存访问来将新元素插入到有序数组中，但是这些访问彼此也非常靠近。只要我们使用各种各样的缓存 (比如 L1，L2，TLB，只要你拥有就行)，这些访问具有的规律性就能弥补大数据量所带来的消耗。

RedBlackTree.insert 的增长率曲线是我们所能够做到的最好的 – 目前没有任何数据结构能解决常规的 SortedSet 的渐进性能问题。但是并不是说没有任何提升空间！

为了让插入更快，我们需要着手于优化我们的实现，通过一些常数因子来加速它。这么做不会改变性能测试曲线的形态，但是能够让曲线在双对数坐标系所处的位置向下平移。

于我们而言，理想的状态是能将 insert 向下平移足够多的距离，让它在全规模的数据上都达到甚至超越 SortedArray 的性能，同时保持对数增长率。

写时复制 (Copy-On-Write) 优化

5

在我们每次向 RedBlackTree 中添加新元素时，都会创建一棵全新的树。新的树会和原来的树共享一些节点，但是在从根节点到新加入的节点的路径上的节点都是新创建的。这种做法可以很 “容易” 地实现值语义，但是会造成一些浪费。

如果树的某些节点没有被其他值引用的话，我们完全可以直接修改它们。这不会造成任何问题，因为根本没有其他人知道这个特定的树的实例。直接修改可以避免绝大部分的复制和内存申请操作，通常这会让性能得到大幅提升。

Swift 通过提供 isKnownUniquelyReferenced 函数来为引用类型实现优化的写时复制值语义，我们已经介绍过相关内容了。但是在 Swift 4 中，语言本身并没有为我们提供为代数数据类型实现写时复制的工具。我们无法访问 Swift 用来包装节点的私有引用类型，因此也就无法获知某个特定节点是不是只有单一引用。(编译器自己还不够聪明，它也并不能帮我们做写时复制优化。) 同时，想要直接获取一个枚举成员里的值，我们也只能先提取一份它的单独的复制。(注意，与此不同，Optional 通过强制解包运算符 !，提供了直接访问存储的值的方式。然而，为我们自己的枚举类型提供类似的原地访问的工具只能被使用在标准库中，在标准库外它们是不可用的。)

所以，为了实现写时复制，我们现在只能放弃我们钟爱的代数数据结构，将所有东西以一种更 “世俗” (或者要我说的话，更**乏味**) 的命令式的形式进行重写，比如使用传统的结构体和类，以及少量的可选值。

基本定义

首先，我们需要定义一个公有结构体，用来表示有序集合。下面的 RedBlackTree2 类型是对一个树节点的引用的简单封装，该节点将作为树的存储根节点。这与 OrderedSet 没有任何不同，所以我们现在对这个模式应该已经相当熟悉了：

```
public struct RedBlackTree2<Element: Comparable>: SortedSet {
  fileprivate var root: Node? = nil

  public init() {}
}
```

接下来，定义树的节点：

```
extension RedBlackTree2 {
  class Node {
```

```
    var color: Color
    var value: Element
    var left: Node? = nil
    var right: Node? = nil
    var mutationCount: Int64 = 0

    init(_ color: Color, _ value: Element, _ left: Node?, _ right: Node?) {
      self.color = color
      self.value = value
      self.left = left
      self.right = right
    }
  }
}
```

在原有的 RedBlackTree.node 枚举成员的基础上，这个类还包含了一个新的属性：mutationCount。它的值表示该节点从被创建以来一共被修改的次数。之后在实现我们的 Collection 时，这个值将被用来构建一种全新的索引方式。(我们这里明确将它定义为 64 位整数，这样就算在 32 位系统中，这个值也不会溢出了。在每个节点中都存储 8 个字节的计数器其实并不太必要，因为我们其实只会使用根节点的这个值。让我们先略过这个细节，这么做能让事情多多少少简单一些，在下一章里我们将会寻找减少浪费的方法。)

不过现在还不是开始说下一章内容的时候！

通过使用不同的类型来代表节点和树，意味着我们可以将节点类型的实现细节隐藏起来，而只将 RedBlackTree2 暴露为 public。对这个集合类型的外部使用者来说，他们将不会把两者混淆起来。在以前，任何人都可以看到 RedBlackTree 的内部实现，都能用 Swift 的枚举字面量语法来创建他们想要的树，这很容易破坏我们的红黑树的特性。

Node 类现在是 RedBlackTree2 结构体的实现细节，将 Node 内嵌在 RedBlackTree2 中很完美地诠释了它们的关系。这也避免了 Node 与同一模块中可能存在的其他同名类型发生命名冲突的问题。同时，这么做还简化了语法：Node 现在将自动继承 RedBlackTree2 的 Element 这一类型参数，我们不再需要明确地进行指定。

> 同样地，按照传统来说，我们只需要一个 bit 的 color 属性，并将它打包到 Node 的引用属性的二进制表示中某个没有在使用的位即可；但是在 Swift 中这么做既不安全，

> 又很麻烦。我们最好还是简单地将 color 保持为一个独立的存储属性，并且让编译器来设置它的存储。

注意，本质上我们将 RedBlackTree 枚举转换为了 Node 类型的可选值引用。.empty 成员现在以 nil 来表示，而非 nil 的值则表示一个 .node。Node 类型是一个在堆上申请内存的引用类型，所以我们将上一个方案中的隐式打包变成了显式的特性，这让我们可以直接访问堆上的引用，并且可以使用 isKnownUniquelyReferenced。

重写简单的查找方法

我们要将算法从代数数据结构重写为适合结构体和类的版本，forEach 就是一个很好的例子。我们通常需要创建两个方法 – 一个用于封装结构体的公有方法，和一个用于节点类型的私有方法。

在树为空的情况下，结构体的方法不需要做任何事。当树不为空时，它只需要将调用传递给树的根节点。我们可以通过可选值链来简洁地表达这个意图：

```
extension RedBlackTree2 {
  public func forEach(_ body: (Element) throws -> Void) rethrows {
    try root?.forEach(body)
  }
}
```

实际上进行中序遍历也只需借助节点的 forEach 方法就可以完成：

```
extension RedBlackTree2.Node {
  func forEach(_ body: (Element) throws -> Void) rethrows {
    try left?.forEach(body)
    try body(value)
    try right?.forEach(body)
  }
}
```

可选值链再一次给我们带来了优雅简洁的代码，太棒了！

我们可以通过使用类似的递归的方式实现 contains，但是现在让我们用一种更 “过程” 式的方法来解决。我们将使用一个 node 变量来在树中导航，每次让当前节点与我们要查找的目标更加接近：

```
extension RedBlackTree2 {
  public func contains(_ element: Element) -> Bool {
    var node = root
    while let n = node {
      if n.value < element {
        node = n.right
      }
      else if n.value > element {
        node = n.left
      }
      else {
        return true
      }
    }
    return false
  }
}
```

这个函数非常类似于我们在第二章中提到的二分查找，不同之处只在于它作用于分支型的引用，而非平面缓冲区 (flat buffer) 的索引。对我们的平衡树来说，该算法同样是对数复杂度。

树形图

显然，我们希望保持酷酷的树形图的特性，所以我们将会重写 diagram 方法。原来的方法作用于 RedBlackTree，而且其类型被转换为了 Optional<Node>。现在我们没法为一个包装了泛型的可选类型定义扩展方法，所以重写该方法最简单的方式是将它转换为一个自由函数：

```
private func diagram<Element>(for node: RedBlackTree2<Element>.Node?, _ top: String =
↪    "", _ root: String = "", _ bottom: String = "") -> String {
  guard let node = node else {
    return root + "•\n"
```

```
    }
    if node.left == nil && node.right == nil {
      return root + "\(node.color.symbol) \(node.value)\n"
    }
    return diagram(for: node.right, top + "    ", top + "┌──", top + "│   ")
      + root + "\(node.color.symbol) \(node.value)\n"
      + diagram(for: node.left, bottom + "│   ", bottom + "└──", bottom + "    ")
}

extension RedBlackTree2: CustomStringConvertible {
  public var description: String {
    return diagram(for: root)
  }
}
```

实现写时复制

我们可以定义辅助方法，来确保在更改任何东西之前，涉及的引用是独立的，这是实现写时复制最简单的方式。对每个存储引用值的属性，我们都需要为其定义一个辅助方法。在我们的例子里有三个引用：RedBlackTree2 结构体中的 root 引用，以及 Node 中的两个子节点引用 (left 和 right)，所以一共需要定义三个方法。

所有的三种情况里，辅助方法都需要检查对应的引用是否唯一，如果不唯一，它需要将其替换为一份复制。对一个节点进行复制，意味着以同样的属性创建一个新节点。下面这个简单的函数可以完成这件事：

```
extension RedBlackTree2.Node {
  func clone() -> Self {
    return .init(color, value, left, right)
  }
}
```

现在让我们来定义这些写时复制辅助方法，先从 RedBlackTree2 的根节点开始：

```
extension RedBlackTree2 {
  fileprivate mutating func makeRootUnique() -> Node? {
```

```
    if root != nil, !isKnownUniquelyReferenced(&root) {
      root = root!.clone()
    }
    return root
  }
}
```

还记得我们的 NSOrderedSet 封装吗？它有一个 makeUnique 方法，做的事情和这里差不多。不过这次我们的引用是可选值，这让事情稍微变复杂了一些。不过好在标准库里有一个接受可选值的 isKnownUniquelyReferenced 的重载方法，所以至少我们不需要为此操心。

当我们在 Swift 中做一些像是实现写时复制或者构建非安全值这种低层级操作时，我们需要特别注意我们代码的语义精确度。拿写时复制为例，我们需要知道和掌握引用计数是在何时，以何种方式发生的改变，这样我们才能避免我们的唯一性检查被临时创建的引用破坏。

举例来说，如果你是经验丰富的 Swift 开发者，可能会忍不住将看起来很笨拙的显式 nil 检查和随后的强制解包替换成漂亮的 let 条件绑定，就像下面这样：

```
if let root = self.root, !isKnownUniquelyReferenced(&root) { // BUG!
  self.root = root.clone()
}
```

这个版本在语法上看起来好一些，**但是它所做的事情完全不同！** let 绑定将作为根节点的新的引用存在，因此 isKnownUniquelyReferenced 将永远返回 false。这会将写时复制优化破坏殆尽。

我们在这里必须如履薄冰，如果有所差池，我们就只能从代码变慢这一点上获得线索，额外的复制并不会造成运行错误，因此很难纠错。

另一方面，如果我们做的复制操作**少于**实际需要，我们的代码就有可能时不时地改变了一个共享的引用类型，从而破坏值语义。与性能上的问题相比较，这个错误要严重得多。如果我们足够幸运，这种对值语义的破坏将会造成运行时错误，比如索引无效等。但是通常结果会更糟：那些拥有引用却被我们忽略的代码有时会导致错误的结果，而我们对于哪里出了问题完全没有线索。这种时候，我只能说，祝你好运！

所以，在实现写时复制的时候一定要**特别特别小心**。即使那些看上去无害的代码美化和改进，都有可能造成非常严重的后果。

让我们回到实现，下面是一个节点中两个子节点的引用的辅助方法。它们都是基于 makeRootUnique 进行简单改编而成：

```
extension RedBlackTree2.Node {
  func makeLeftUnique() -> RedBlackTree2<Element>.Node? {
    if left != nil, !isKnownUniquelyReferenced(&left) {
      left = left!.clone()
    }
    return left
  }

  func makeRightUnique() -> RedBlackTree2<Element>.Node? {
    if right != nil, !isKnownUniquelyReferenced(&right) {
      right = right!.clone()
    }
    return right
  }
}
```

插入

现在我们已经准备好重写 insert 了。和之前一样，我们依然需要将它分解为两个部分：一个针对封装结构体的公有方法，以及一个针对节点类的私有方法。(insert 已经被拆分为若干更小的负责各自独立子任务的部分，所以这里主要涉及的是每个部分放到哪里的问题。)

封装方法相对简单。它是一个可变方法，所以它必须调用 makeRootUnique 来确保变更根节点是可行的。如果树是空的，我们需要为被插入的元素创建一个新的根节点；如果树不为空，我们将调用传递给已经存在的根节点，这个时候我们已经可以确保根节点的引用已经是唯一的了：

```
extension RedBlackTree2 {
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪  Element) {
    guard let root = makeRootUnique() else {
      self.root = Node(.black, element, nil, nil)
      return (true, element)
```

```
    }
    defer { root.color = .black }
    return root.insert(element)
  }
}
```

我们使用 defer 来无条件地将根节点染黑，这样可以确保满足红黑树的第一个要求。

节点的 insert 对应我们原来的 RedBlackTree 中的老朋友，_inserting 方法：

```
extension RedBlackTree2.Node {
  func insert(_ element: Element)  -> (inserted: Bool, memberAfterInsert: Element) {
    mutationCount += 1
    if element < self.value {
      if let next = makeLeftUnique() {
        let result = next.insert(element)
        if result.inserted { self.balance() }
        return result
      }
      else {
        self.left = .init(.red, element, nil, nil)
        return (inserted: true, memberAfterInsert: element)
      }
    }
    if element > self.value {
      if let next = makeRightUnique() {
        let result = next.insert(element)
        if result.inserted { self.balance() }
        return result
      }
      else {
        self.right = .init(.red, element, nil, nil)
        return (inserted: true, memberAfterInsert: element)
      }
    }
    return (inserted: false, memberAfterInsert: self.value)
  }
}
```

这里比较讨厌的是，我们只能把 switch 转换为一个级联的 if 操作，并且加入一堆用来更改值的语句。但是它还是和原来的代码具有相同的结构。事实上，我想要是 Swift 编译器能够自动帮我们进行这个重写就好了。(代数数据结构的写时复制自动优化对编译器来说将会是一个长足的进步，我们也许在**将来**，比如 Swift 10 的时候就能见到这个特性吧...)

回到现实吧，我们现在来看看如何平衡这棵树。

RedBlackTree 中我们用了一种基于模式匹配的方式，针对红黑树平衡给出了一种令人难忘的巧妙实现。在文明社会的现代编码中，模式匹配算得上一种优雅的武器。它是如此优美，如此绚烂！

但是我们现在只能用一大坨充满恶臭的赋值语句来玷污这种优雅了：

```
extension RedBlackTree2.Node {
  func balance() {
    if self.color == .red  { return }
    if left?.color == .red {
      if left?.left?.color == .red {
        let l = left!
        let ll = l.left!
        swap(&self.value, &l.value)
        (self.left, l.left, l.right, self.right) = (ll, l.right, self.right, l)
        self.color = .red
        l.color = .black
        ll.color = .black
        return
      }
      if left?.right?.color == .red {
        let l = left!
        let lr = l.right!
        swap(&self.value, &lr.value)
        (l.right, lr.left, lr.right, self.right) = (lr.left, lr.right, self.right, lr)
        self.color = .red
        l.color = .black
        lr.color = .black
        return
      }
    }
```

```
    if right?.color == .red {
      if right?.left?.color == .red {
        let r = right!
        let rl = r.left!
        swap(&self.value, &rl.value)
        (self.left, rl.left, rl.right, r.left) = (rl, self.left, rl.left, rl.right)
        self.color = .red
        r.color = .black
        rl.color = .black
        return
      }
      if right?.right?.color == .red {
        let r = right!
        let rr = r.right!
        swap(&self.value, &r.value)
        (self.left, r.left, r.right, self.right) = (r, self.left, r.left, rr)
        self.color = .red
        r.color = .black
        rr.color = .black
        return
      }
    }
  }
}
```

没错，这就是我们原来的平衡函数直接用命令式的方式重写变形后的样子。这种代码真是闻者伤心，见者落泪。

我们可以稍微重构一下这些代码，让它看起来更顺眼一些 (同时更快一些)。比如，我们可以将 balance 嵌入到 insert 函数里，这样我们可以根据插入新元素的左右方向，修剪掉一些不必要的分支。

不过更简单的是，打开一本数据结构的教材，然后把里面关于红黑树的插入的算法直接照搬到 Swift 里。我们在这里不会去做这件事，因为这实在太无聊了，而且也不会让我们的代码变快很多。

> 如果你依然对此感兴趣，可以看看我为 Swift 2 设计的红黑树实现。我没有将它升级到 Swift 3 或 4，这是因为我们在接下来的章节要介绍的 B 树要比它好得多，所以我也就没有动力去升级了。

实现 Collection

我们来看看这次能不能给出一个更好的索引实现吧。

在 RedBlackTree 里，我们使用了一个对元素值进行包装的 dummy 索引类型。为了实现 index(after:)，我们需要从头开始寻找元素在树中的位置，这个操作的时间复杂度是 $O(\log n)$。

也就是说，当我们在整个 RedBlackTree 中用索引进行迭代的时候，我们要做的是一个 $O(n \log n)$ 的操作，这里 n 是集合类型中元素的个数。这是很糟糕的结果，在一个集合类型中迭代所有元素应该只需要 $O(n)$ 的时间才对！(虽然这实际上不是 Collection 的要求，但是这绝对是一个合理的预期。)

那么，我们怎么办呢？一种加速的办法是让索引包含从根到某个元素的整个路径。

不过，这个想法有一个小问题，索引不应该持有集合类型中某些部分的强引用。所以，我们可以用一个弱引用的数组来表示路径。

索引定义

Swift 4 里，我们不能直接定义一个弱引用数组，所以首先我们需要为弱引用定义一个简单的结构体封装：

```
private struct Weak<Wrapped: AnyObject> {
  weak var value: Wrapped?

  init(_ value: Wrapped) {
    self.value = value
  }
}
```

我们现在可以把路径声明为一个包含 Weak 值的数组了。在使用时，我们需要记住在 Weak 值后添加 .value 才能取出实际的引用 (比如 path[0].value)，不过这只是一个存在于表面的小问题。这个封装结构体不会在运行时造成任何性能问题。不过每次都这么做还是有一点麻烦，在语言层面添加 weak 或是 unowned 的数组支持也许是个不错的 Swift 特性提案。不过我们这里要做的是实现一个有序集合，而不是增强 Swift 本身，所以让我们先接受这一点语言上的小瑕疵并且继续前行吧。

现在，我们可以定义实际的索引类型了。基本上来说，它是一个对路径的封装，里面保存了一个以弱引用方式持有节点的数组。为了能够简单地验证索引，我们还添加了一个直接指向根节点的弱引用，以及在索引被创建时根节点被更改次数的复制：

```
extension RedBlackTree2 {
  public struct Index {
    fileprivate weak var root: Node?
    fileprivate let mutationCount: Int64?

    fileprivate var path: [Weak<Node>]

    fileprivate init(root: Node?, path: [Weak<Node>]) {
      self.root = root
      self.mutationCount = root?.mutationCount
      self.path = path
    }
  }
}
```

图 5.1 描述了一个索引的例子，它指向的是红黑树里的第一个值。注意，索引使用的都是弱引用，所以它不会保留任何一个树的节点。isKnownUniquelyReferenced 函数只计算强引用，这样一来，特定索引就不会影响到树的原地变更了。不过，这也意味着变更将会破坏之前创建的索引中的节点引用，所以我们需要对索引验证格外小心。

起始索引和结束索引

让我们开始写 Collection 的相关方法吧。

首先，我们可以将 endIndex 定义为一个空路径，这很简单：

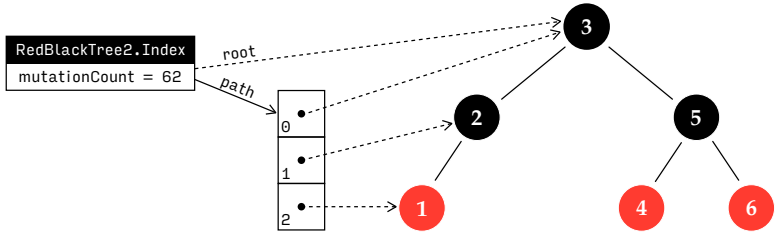


图 5.1: 由 RedBlackTree2.Index 实现的一棵简单红黑树的起始索引。虚线表示弱引用。

```
extension RedBlackTree2 {
  public var endIndex: Index {
    return Index(root: root, path: [])
  }
```

起始索引也不太困难，我们只需构建一条通往树里最左侧元素的路径：

```
  // - 复杂度: O(log(n)); 这违背了 `Collection` 的要求。
  public var startIndex: Index {
    var path: [Weak<Node>] = []
    var node = root
    while let n = node {
      path.append(Weak(n))
      node = n.left
    }
    return Index(root: root, path: path)
  }
}
```

该操作会消耗 $O(\log n)$ 的时间，这再一次打破了 Collection 的相应要求。当然，我们有办法可以修复它，但是相比在 RedBlackTree 里，这需要更多的技巧，况且我们也无法确认收益是否能值回这样的付出，所以我们暂时先搁置这个问题。如果在一个实际的软件包里，我们可能需要在文档中指出这一点，并给出警告，让我们的用户知道这件事情。(有些集合类型操作可能会因此变慢很多。)

索引验证

接下来我们想讨论一下索引验证。对 RedBlackTree2 来说，我们需要在每次发生变更后让所有的索引失效，因为变更有可能修改或者替换了存储于索引路径上的某些节点。

要做到这一点，我们需要验证树的根节点和索引中的根节点引用是否相同，然后再检查它们是否拥有相同的变更次数。对一个空树的有效索引来说，其根节点为 nil，这让事情稍微复杂了一些：

```
extension RedBlackTree2.Index {
  fileprivate func isValid(for root: RedBlackTree2<Element>.Node?) -> Bool {
    return self.root === root
      && self.mutationCount == root?.mutationCount
  }
}
```

如果这个函数没有出错，我们就知道索引属于正确的树，并且树并没有在索引创建后被变更。也就是说，索引路径包含的是有效的引用。否则，就意味着路径上包含的弱引用或者节点里有变更后的值，因此我们也就不能再使用这个索引。

要进行索引比较，我们需要检查两个索引是否属于同一棵树，以及它们对于这棵树来说是否都是有效的。这里有一个静态函数做这件事情：

```
extension RedBlackTree2.Index {
  fileprivate static func validate(_ left: RedBlackTree2<Element>.Index, _ right:
  ↪  RedBlackTree2<Element>.Index) -> Bool {
    return left.root === right.root
      && left.mutationCount == right.mutationCount
      && left.mutationCount == left.root?.mutationCount
  }
}
```

下标

通过索引来实现下标操作非常激动人心，因为它包含了不止一个，而是两个感叹号!!：

```
extension RedBlackTree2 {
  public subscript(_ index: Index) -> Element {
    precondition(index.isValid(for: root))
    return index.path.last!.value!.value
  }
}
```

这里对于强制解包的使用是没有问题的，通过 endIndex，或者是一个无效的索引来进行下标操作，确实应该造成崩溃。我们可以提供更好的错误信息，但是现在来说这样就足够了。

索引比较

索引必须可以被比较，要实现这一点，我们采用和 RedBlackTree 里一样的做法，简单地将两个索引路径中的最后一个节点取出来，然后比较它们的值。现在让我们来看看可以如何取得一个索引的当前节点：

```
extension RedBlackTree2.Index {
  /// 前置条件：`self` 是有效索引。
  fileprivate var current: RedBlackTree2<Element>.Node? {
    guard let ref = path.last else { return nil }
    return ref.value!
  }
}
```

这个属性假设了索引是有效的，否则强制解包可能会失败，并让我们的程序崩溃。

我们现在可以写比较的代码了，这和我们之前做的很相似：

```
extension RedBlackTree2.Index: Comparable {
  public static func ==(left: RedBlackTree2<Element>.Index, right:
  ↪  RedBlackTree2<Element>.Index) -> Bool {
    precondition(RedBlackTree2<Element>.Index.validate(left, right))
    return left.current === right.current
  }
```

```
  public static func <(left: RedBlackTree2<Element>.Index, right:
  ↪  RedBlackTree2<Element>.Index) -> Bool {
    precondition(RedBlackTree2<Element>.Index.validate(left, right))
    switch (left.current, right.current) {
    case let (a?, b?):
      return a.value < b.value
    case (nil, _):
      return false
    default:
      return true
    }
  }
}
```

索引步进

最后，我们要实现索引的步进操作。我们将把实际的工作转交给索引类型上的一个可变方法，我们之后马上会去实现这个方法：

```
extension RedBlackTree2 {
  public func formIndex(after index: inout Index) {
    precondition(index.isValid(for: root))
    index.formSuccessor()
  }

  public func index(after index: Index) -> Index {
    var result = index
    self.formIndex(after: &result)
    return result
  }
}
```

想要找到一个已存在索引的后续索引，我们需要寻找当前节点的右子树中的最左侧节点。如果当前节点没有右子树，那么我们就回到向上一级的节点，并判断当前节点是否存在于这个上级节点的**左**子树中。

如果树里没有这样的上级节点，那就意味着我们到达了集合类型的尾部。这种情况下，返回一个和集合类型的 endIndex 一样的空路径索引正是我们所需要的：

```
extension RedBlackTree2.Index {
  /// 前置条件: 除了 `endIndex` 以外，`self` 也是有效索引。
  mutating func formSuccessor() {
    guard let node = current else { preconditionFailure() }
    if var n = node.right {
      path.append(Weak(n))
      while let next = n.left {
        path.append(Weak(next))
        n = next
      }
    }
    else {
      path.removeLast()
      var n = node
      while let parent = self.current {
        if parent.left === n { return }
        n = parent
        path.removeLast()
      }
    }
  }
}
```

为了满足 BidirectionalCollection，我们还需要实现从任意索引开始向前返回。实现的代码和 index(before:) 具有完全相同的结构，只是我们需要将左右互换，另外，我们还需要对 endIndex 做特殊处理：

```
extension RedBlackTree2 {
  public func formIndex(before index: inout Index) {
    precondition(index.isValid(for: root))
    index.formPredecessor()
  }

  public func index(before index: Index) -> Index {
```

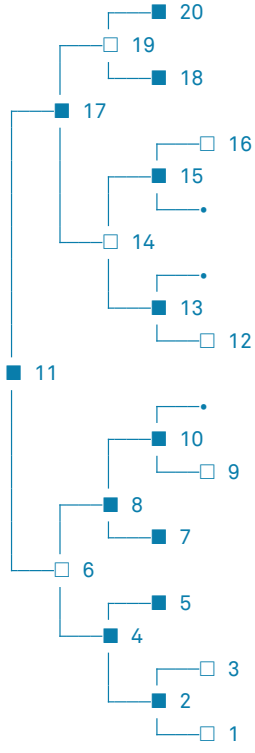
```
    var result = index
    self.formIndex(before: &result)
    return result
  }
}

extension RedBlackTree2.Index {
  /// 前置条件: 除了 `startIndex` 以外，`self` 也是有效索引。
  mutating func formPredecessor() {
    let current = self.current
    precondition(current != nil || root != nil)
    if var n = (current == nil ? root : current!.left) {
      path.append(Weak(n))
      while let next = n.right {
        path.append(Weak(next))
        n = next
      }
    }
    else {
      path.removeLast()
      var n = current
      while let parent = self.current {
        if parent.right === n { return }
        n = parent
        path.removeLast()
      }
    }
  }
}
```

例子

呼，全部搞定了！让我们试试看这个新的类型吧：

```
var set = RedBlackTree2<Int>()
for i in (1 ... 20).shuffled() {
```

```
    set.insert(i)
}
▶ set
```




```
▶ set.contains(13)
  true

▶ set.contains(42)
  false

▶ set.filter { $0 % 2 == 0 }
  [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

性能测试

在 RedBlackTree2 上进行通常的性能测试，得到下面的结果图 5.2。insert、contains 和 forEach 的曲线形状与我们在 RedBlackTree 中看到的结果大致一样。但是 for-in 的性能看起来却非常奇怪！

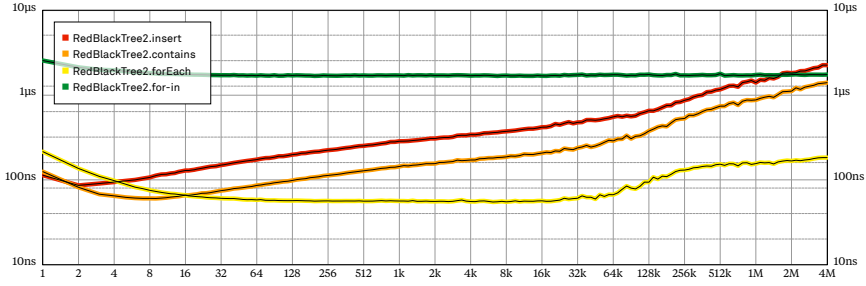


图 5.2: RedBlackTree2 操作的性能测试，单次迭代的情况下，输入元素个数和平均执行时间的对数关系图。

优化迭代性能

我们重新设计了索引类型，这样 index(after:) 在平摊后将以常数时间运行。看起来我们成功了：for-in 的曲线很平缓，它只在集合尺寸较大时有些许增加，但是它在整张图里实在是太高了！对于少于 100,000 个元素的小集合来说，显然插入一个元素比迭代还要快，这是不合常理的。

图 5.3 将 RedBlackTree 和 RedBlackTree2 的 for-in 性能进行了对比。我们可以看到，RedBlackTree2.index(after:) 平摊下来确实是一个常数时间复杂度的操作，对于非常庞大的数据集，曲线也保持了平坦。而 RedBlackTree 的操作中索引步进是对数复杂度，它的渐进特性 (随着数据集扩大的耗时) 要差一些。不过因为这条曲线的起点要低得多，所以在元素数量达到大约一千六百万之前，都要比 RedBlackTree2 更快，这个值已经远远超过我们的性能测试图的显示范围了。这样的常数复杂度可以说不要也罢。

想要让 RedBlackTree2 和 RedBlackTree 里的原始的索引有相同竞争力的话，我们需要将它的索引操作提速四倍。虽然这应该是可能的，但是现在我们还不清楚要如何才能达到这个目标。不管那么多，我们可以先来尝试一下！

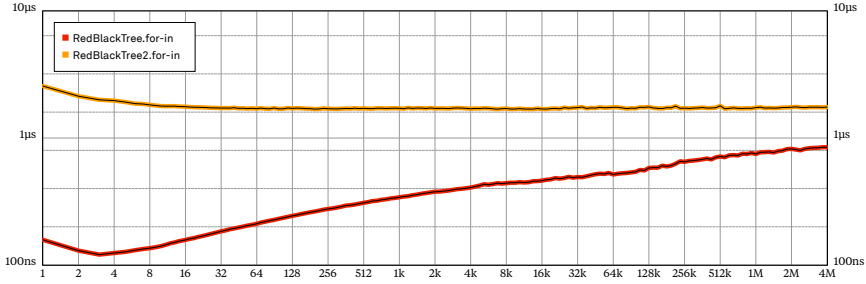


图 5.3: 基于树结构的有序集合中 **for-in** 实现的性能对比。

移除索引验证是一个努力方向，因为它是迭代的主要开销来源之一。将索引验证完全移除显然不是什么好主意，但是在 for-in 循环的这个特例里，索引验证是完全没有必要的：集合在迭代过程中保持不变，所以验证永远不会失败。想要移除验证，我们可以放弃 Collection 默认的 IndexingIterator，取而代之，改用特殊的 IteratorProtocol 实现，它可以直接处理索引：

```
extension RedBlackTree2 {
  public struct Iterator: IteratorProtocol {
    let tree: RedBlackTree2
    var index: RedBlackTree2.Index

    init(_ tree: RedBlackTree2) {
      self.tree = tree
      self.index = tree.startIndex
    }

    public mutating func next() -> Element? {
      if index.path.isEmpty { return nil }
      defer { index.formSuccessor() }
      return index.path.last!.value!.value
    }
  }
}

extension RedBlackTree2 {
  public func makeIterator() -> Iterator {
    return Iterator(self)
  }
```

}

注意，很明显现在迭代器包含了一份对树的冗余复制。虽然我们在 next 中从未使用过这个存储属性，但是它扮演的角色却非常重要：它为我们保持着根节点，这保证了树在迭代器的生命周期内不会被释放。

通过这个自定义的迭代器，我们的 for-in 得到了两倍的性能提升。(参见 图 5.4 中的 for-in2 曲线。) 这是一个不错的结果，但是我们想要 400% 的提速，所以我们来看看还有哪些方法能让迭代速度更快。

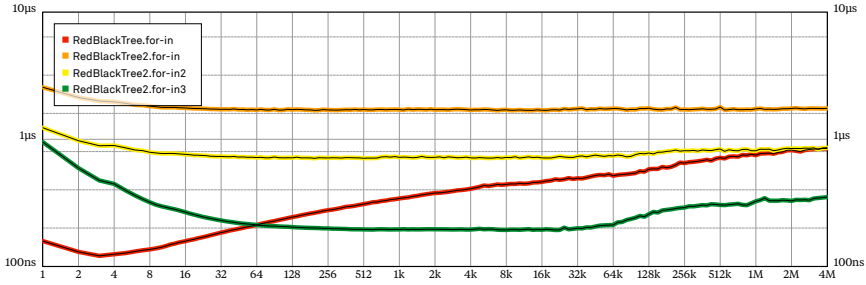


图 5.4: 基于树结构的有序集合中 for-in 实现的性能对比。RedBlackTree2.for-in2 通过实现自定义迭代器移除了不必要的索引验证步骤，for-in3 将弱引用替换为不安全的 unowned 引用。

一个值得注意的关键点是，我们对更改计数进行的检查，确保了一个有效索引的路径上永远不会有过期的弱引用，因为路径所关联的树还存在，所以那些节点和索引被创建时的节点完全是同样的东西。也就是说，我们可以安全地将路径上的弱引用替换为 unowned(unsafe)。这样一来，所有的引用计数的管理操作都被移除了，这带来了另一次两倍的性能提升。(参见上图中的 for-in3 曲线)。这让我们获得了可以接受的 for-in 性能，所以我们的优化可以在此告一段落了。

在 RedBlackTree2 中，我们选择了渐进性能更好的索引算法，但是它并没有直接为我们带来更快的结果：算法的优势受累于实现的细节。然而，我们成功地进行了优化，使这个 “更好” 的算法的结果可以匹敌于 “更差” 的算法。在这个例子中，我们非常幸运，在两个相对简单的优化步骤之后就达到了满意的效果。一般来说，我们需要做的远不止如此！

插入性能

RedBlackTree2 为我们带来了一个非常高效的基于树结构的 SortedSet 实现。如 图 5.5 所示，通过原地变更，插入性能有了 300–500% 的提升。RedBlackTree2.insert 不再迟钝，现在它能在仅仅 12 秒内就完成四百万次插入操作。

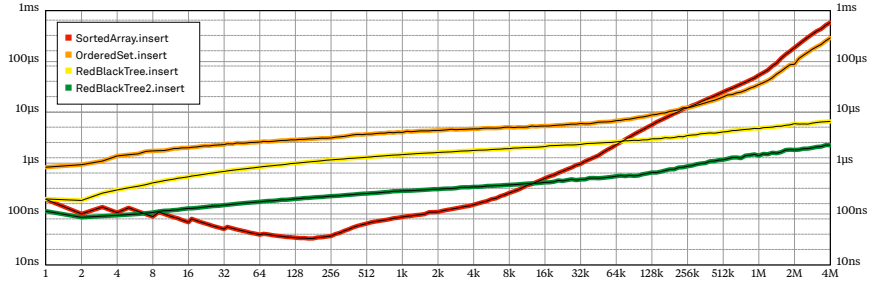


图 5.5: 对比四种 insert 实现的性能。

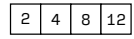
但是在低于 8,000 个元素的时候，它还是比不上 SortedArray：后者要比红黑树快上四倍。嗯...虽然肯定有办法将红黑树的插入操作做进一步优化，但是看起来可能达不到 400% 的加速。那么，下面我们要做些什么呢？

B 树

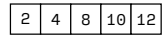
6

小尺寸 SortedArray 的原生性能看起来几乎不可能挑战。所以我们也就不再争取了，不如另辟蹊径，尝试通过这些高性能的小尺寸有序数组来构建一个有序集合！

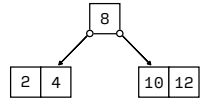
一开始非常简单：我们只需要将元素插入到单个数组，直到达到预先规定的元素上限。比如说我们想要将数组的长度保持在四以内，而现在已经插入了四个元素：



如果我们此时需要插入 10，那么数组长度就会超过规定：

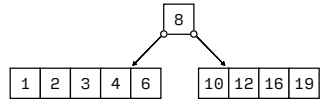


我们需要采取一些行动来避免这样的事情发生。一种选择是将数组分割为两半，然后使用中间的那个元素作为两边的分隔元素：

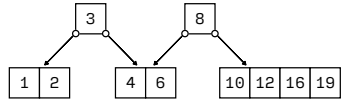


现在，我们有了一个小巧的树形结构，它的叶子节点包含了小尺寸的有序数组。看起来这是一种很有前途的方式：它将有序数组和搜索树合并到了一个统合的数据结构中，这有希望给在小尺寸时给我们带来数组那样超级快的插入操作，同时在大数据集中保持树那样的对数性能。

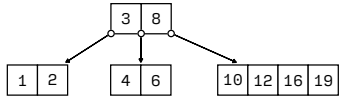
让我们看看如果继续添加更多元素会发生什么。我们将继续向两个叶子数组中添加新值，直到某一个再次超过范围：



当这种情况发生时，我们需要再进行一次分割，这将给我们带来三个数组以及两个分隔的元素：

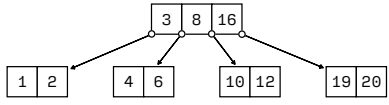


对这两个分隔的值，我们应该做些什么呢？我们已经将其他所有元素放在有序数组中了，所以看起来把分隔值也放到它们自己的有序数组里会是个明智的选择：

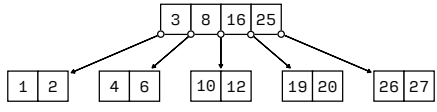


很显然，新的混合数组树中的每个节点都持有一个小尺寸有序数组。到目前为止，我很喜欢这个想法，它优雅而且一致。

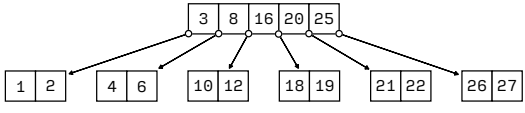
接下来，如果我们想要插入 20 这个值的话，会怎么样呢？它会被放到最右侧的数组中去，不过那里已经有四个元素了，所以我们需要再进行一次分割，将中间值 16 提取出来，成为新的分隔元素。小菜一碟，我们只要将它插入到顶端的数组里就行了：



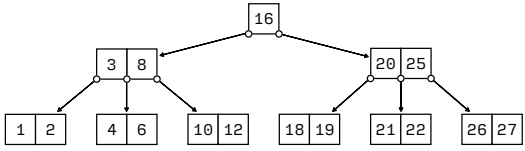
很好，让我们继续，在插入 25、26 和 27 之后，最右侧的数组又溢出了，于是我们再次提取新的分隔元素，这次是 25:



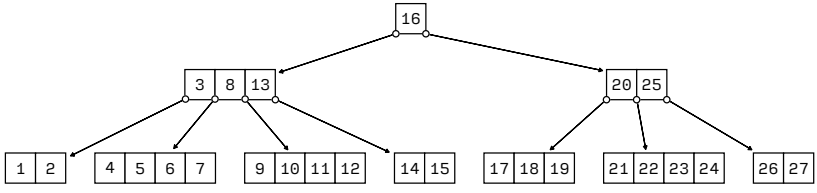
不过，现在顶端的数组也满了。如果我们接着插入 18、21 和 22，等待我们的便是下面的情况：



接下来怎么办呢？我们可不能放任这个分隔数组就这样膨胀下去。之前，我们通过分割溢出的数组来解决这个问题，这里我们完全可以如法炮制。



哈，完美：通过分割第二层的数组，我们可以在树上添加第三层数组。这让我们可以无限地添加新的元素，当第三层的数组被填满时，我们将添加第四层，以此类推，以至无穷：



我们刚刚发明了一种全新的数据结构！这是历史性的时刻！

但是高兴得太早了，其实 Rudolf Bayer 和 Ed McCreight 早在 1971 年就提出过一样的想法，他们将这个发明命名为 **B 树**。真是晴天霹雳，我花了一整本书来介绍一个东西，但是你却告诉我这玩意半个世纪之前就有了，简直悲剧。

有趣的事实是：红黑树实际上是在 1978 年衍生出来的一种 B 树的特殊形式。这些数据结构都经历了岁月的洗礼。

B 树的特性

正如我们所见，**B 树**和红黑树一样，都是搜索树，但是它们的结构却有所不同。在 B 树中，节点可能会拥有成百甚至上千的子节点，而不仅仅是两个。不过子节点的个数也并非完全没有限制：节点数肯定会落在某个范围内。

每个节点的最大子节点数在树创建的时候就已经决定了，这个值被叫做 B 树的 **“阶”**。(阶的英文和顺序一样，都是 order，但是阶和元素的顺序无关。) 注意，节点中能存放的元素的最大数量要比它的阶的数字小一。这很可能会导致计算索引时发生差一错误，所以当我们在实现和使用 B 树的时候，一定要将此铭记于心。

在上面，我们构建了一棵阶为 5 的 B 树。实际运用中，阶通常介于 100 到 2,000 之间，5 可以说是小的非同寻常。不过，有 1,000 个子节点的节点没法在页面上表示出来，使用一个能画出来的例子能让我们更容易地理解 B 树。

为了能在树中定位元素，每个内部节点在它的两个子节点之间存储一个元素，这和红黑树中值存储在左右子树之间是类似的。也就是说，一个含有 1,000 个子节点的 B 树节点将存储 999 个按照升序排列好的值。

为了保持查找操作的高效，B 树需要维护如下平衡：

1. **最大尺寸：**每个节点最多存储 order - 1 个按照升序排列的元素。
2. **最小尺寸：**非根节点中至少要填满一半元素，也就是说，除了根节点以外，其余每个节点中的元素个数至少为 (order - 1) / 2。
3. **深度均匀：**所有叶子节点在树中所处的深度必须相同，也就是位于从顶端根节点向下计数的同一层上。

要注意的是，后两个特性是我们的插入方式所诱发的自然结果；我们不需要做任何额外的工作，就可以保证节点不会变得太小，并且所有的叶子都在同一层上。(在其他 B 树操作中，想要维持这些特性会困难的多。比如，删除一个元素可能导致出现不满足要求的节点，此时需要对它进行修正。)

根据这些规则，一个阶为 1,000 的 B 树的节点所能够持有的元素个数在 499 至 999 之间 (包括 999)。唯一的例外是根节点，它不受下限的限制：根节点中可以包含 0 到 999 个元素。(也正因如此，我们才能创建一棵元素个数少于 499 的 B 树。) 这样一来，单独一个 B 树的节点中能够持有的元素个数和一棵 **10 到 20 层**深的红黑树所能持有的元素个数相当！

将如此之多的元素存放在单个节点中有两个主要好处：

1. **降低内存开销。**红黑树中的每个值都存储在一个独立申请于堆上的节点中，该节点还包括了一个指向方法查找表的指针，自身引用计数，以及两个分别指向左右子节点的引用。而 B 树中的节点将批量存储元素，这可以明显地降低内存申请的开销。(具体节省了多少取决于元素个数以及树的阶。)
2. **存取模式更适合内存缓存。** B 树将元素存储在小的连续缓冲区中。如果缓冲区的尺寸经过精心设计，能够全部载入 CPU 的 L1 缓存的话，对它们的操作将会明显快于等效代码对红黑树进行的操作，因为红黑树中的值是散落在堆上的。

为 B 树添加额外的一层，可以使 B 树中所能存储的最大元素个数以阶的乘积的方式增加 (新的最大元素个数 = 原最大个数 × 树的阶)，B 树的稠密特性可见一斑。比如，对于一个阶为 1,000 的 B 树，其最小和最大元素个数随着树的层数的增长情况如下：

Depth	Minimum size	Maximum size
1	0	999
2	999	999 999
3	499 999	999 999 999
4	249 999 999	999 999 999 999
5	124 999 999 999	999 999 999 999 999
6	62 499 999 999 999	999 999 999 999 999 999
⋮	⋮	⋮
n	$2\left\lceil \frac{order}{2} \right\rceil^{n-1} - 1$	$order^n - 1$

很显然，我们几乎不太会有机会处理层级较多的 B 树。理论上，B 树的深度是 $O(\log n)$，这里 n 是元素的个数。但是这个对数的底数实在太大，在真实世界的计算机中，由于可用内存数量的限制，实际上对于任意我们可预期的输入尺寸，可以说 B 树的深度都是 $O(1)$ 级别的。

> 最后一句话看起来好像很有道理，而且都这么想的话，会让人觉得是不是只要把时间尺度放大到一个人的剩余生命的话，在实践中所有的算法就都是 $O(1)$ 复杂度的了。显然，我不会觉得一个跑到我死都没完成的算法是可以 “实践” 的。不过，你确实能将整个宇宙中可以观测到的粒子都放到一个阶为 1,000 的 B 树中，而这棵 B 树也不过就是 30 多层。所以千万不要去和对数较真，这没什么意义。

在 B 树中，**绝大多数**元素都是存储在叶子节点中的，这是 B 树如此巨大的扇出 (fan-out) 所导致的另一个有趣的结果。在一个阶为 1,000 的 B 树中，至少 99.8% 的元素存在于叶子节点中。因此，在批量操作 B 树元素 (比如迭代) 时，我们大多数时候需要将注意力放在叶子节点上，对叶子节点进行优化，保持处理迅速：通常，在性能测试的结果中，花在中间节点的时间甚至都不做记录。

奇怪的是，除此之外，B 树的节点数和它的元素数理论上还是成比例的。大多数的 B 树算法和对应的二叉树代码具有相同级别的时间复杂度。不过，在简化后的时间复杂度的背后，实践中复杂度的常数因子也很重要，B 树所做的正是在常数因子上进行优化。不要对此感到意外，因为如果我们完全不关心常数因子的话，这本书在讲完 RedBlackTree 之后就可以终结了！

基本定义

说得足够了，让我们开始动手吧！

和 RedblackTree2 一样，我们通过为根节点引用定义一个公有的封装结构体开始，来实现 B 树：

```
public struct BTree<Element: Comparable> {
  fileprivate var root: Node

  init(order: Int) {
    self.root = Node(order: order)
  }
}
```

在 RedblackTree2 中，root 是一个可选引用，这样空树就不需要在内存中申请任何东西。不过将根节点定义为非可选值，将使我们的代码变得短一些。在这里，我推崇简洁至上。

节点类需要持有两个缓冲区：一个用来存储元素，另一个用来存储子节点的引用。最简单的方法是使用数组，所以就让我们从数组开始。

为了让我们的树更易于测试，树的阶将不是一个在编译时就决定的常数，我们会通过上面的初始化方法自定义树的阶。我们将阶数以只读属性的方式存储在每个节点中，这样我们就能轻易地获取它们了：

```
extension BTree {
  final class Node {
    let order: Int
    var mutationCount: Int64 = 0
    var elements: [Element] = []
    var children: [Node] = []

    init(order: Int) {
      self.order = order
    }
  }
}
```

和前一章一样，我们为节点添加了 mutationCount 属性。如前所述，现在在每个节点中存储变更计数值的浪费要少得多了，一个典型的节点将会存储数千字节的数据，所以再为它额外添加 8 个字节也无关紧要：

图 6.1 显示了我们例子中的 B 树以 BTree 表示的方式。注意在 elements 和 children 数组中的索引的组织方式：对于 0 ..< elements.count 中的任意一个 i，elements[i] 的值要比 children[i] 中的任意值都要大，但是比 children[i + 1] 中的任意值都要小。

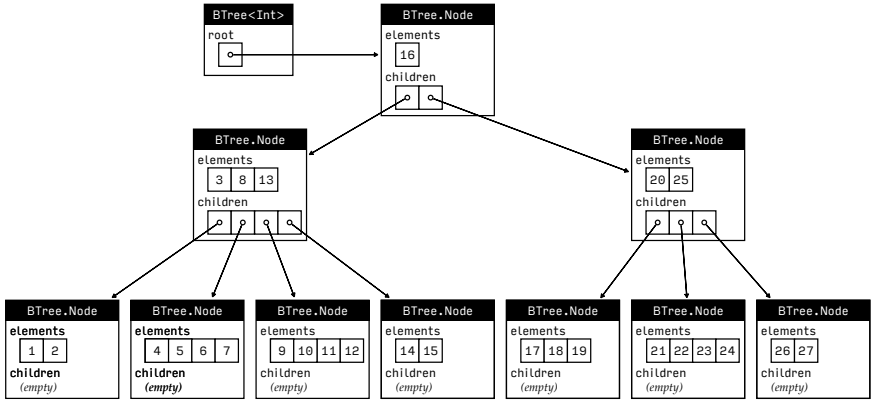


图 6.1: 以 BTree 表示的 95 页的 B 树。

默认初始化方法

用户可以自定义树的阶，但是为了实现 SortedSet，我们还需要一个无参初始化方法；我们应该用什么值来做默认的阶呢？

一种选择是直接在代码中设定一个具体的值：

```
extension BTree {
  public init() { self.init(order: 1024) }
}
```

不过，我们还能做得更好。性能测试表明，当元素缓冲区的最大尺寸大约为 CPU L1 缓存尺寸的四分之一时，B 树拥有最快的速度。Darwin 系统 (Apple 操作系统底层的内核) 提供了 sysctl 指令用来查询缓存的大小，Linux 下也有对应的 sysconf 指令。在 Swift 中，我们可以这样来访问它们：

```
#if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
import Darwin
#elseif os(Linux)
import Glibc
#endif

public let cacheSize: Int? = {
  #if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
    var result: Int = 0
    var size = MemoryLayout<Int>.size
    let status = sysctlbyname("hw.l1dcachesize", &result, &size, nil, 0)
    guard status != -1 else { return nil }
    return result
  #elseif os(Linux)
    let result = sysconf(Int32(_SC_LEVEL1_DCACHE_SIZE))
    guard result != -1 else { return nil }
    return result
  #else
    return nil // 未知平台
  #endif
}()
```

> 在 Darwin 中，还有很多其他的 sysctl 名称；你可以通过在命令行窗口运行 man 3 sysctl 来获取最常用的名称列表。另外，sysctl -a 可以让你获取一份所有可用查询以及它们的当前值的列表，confnames.h 列出了所有你能用来作为 sysconf 参数的符号名称。

当知道缓存尺寸后，配合上标准库中的 MemoryLayout 获取单个元素在连续内存缓冲区中占用的字节数，我们就可以定义无参的初始化方法了：

```
extension BTree {
```

```
  public init() {
    let order = (cacheSize ?? 32768) / (4 * MemoryLayout<Element>.stride)
    self.init(order: Swift.max(16, order))
  }
}
```

如果无法确定缓存尺寸，则使用一个看上去合理的默认数值。max 的调用确保了即使在元素尺寸巨大的时候，我们也还是能得到一棵足够茂密的树。

使用 forEach 迭代

让我们来看看 B 树中的 forEach 方法吧。和之前一样，封装结构体的 forEach 方法只是简单地将调用传递给根节点：

```
extension BTree {
  public func forEach(_ body: (Element) throws -> Void) rethrows {
    try root.forEach(body)
  }
}
```

但我们要怎样才能访问到一个节点中的所有元素呢？如果该节点是一个叶子节点，那我们只需要在它的所有元素上调用 body 就行了。如果一个节点拥有子节点，那我们就需要在访问元素的时候，以递归调用的方式将对子节点的访问一个个插入其中：

```
extension BTree.Node {
  func forEach(_ body: (Element) throws -> Void) rethrows {
    if children.isEmpty {
      try elements.forEach(body)
    }
    else {
      for i in 0 ..< elements.count {
        try children[i].forEach(body)
        try body(elements[i])
      }
      try children[elements.count].forEach(body)
```

```
    }
  }
}
```

注意，第一个子节点的元素要比当前节点的第一个元素要小，所以我们需要以一个递归调用开头。同时，在最后一个元素之后，还存在一个子节点，其中的元素都比该末尾元素要大，我们还需要在最后对它加上一次额外的调用。

查找方法

为了在 B 树中查找某个特定元素，我们首先要写一个用于在某个节点内部查找元素的工具方法。

因为节点中的元素数组其实是已排序的，所以这项任务就简化为了实现一个与 SortedArray 相同的二分查找。不过为了让事情更简单一些，这次我们的函数还集成了对最终找到的元素进行成员测试的部分：

```
extension BTree.Node {
  internal func slot(of element: Element) -> (match: Bool, index: Int) {
    var start = 0
    var end = elements.count
    while start < end {
      let mid = start + (end - start) / 2
      if elements[mid] < element {
        start = mid + 1
      }
      else {
        end = mid
      }
    }
    let match = start < elements.count && elements[start] == element
    return (match, start)
  }
}
```

这里我们将一个节点中的索引称作**位置** (slots)，来和稍后定义的集合类型的索引进行区分。(在节点内部的数组中，包括 elements 和 children，我们都将用位置来代指其索引。)

现在我们有了计算位置的方法，在实现 contains 的时候，我们会需要这个方法。同样，我们将封装结构体上的 contains 调用转发给根节点：

```
extension BTree {
  public func contains(_ element: Element) -> Bool {
    return root.contains(element)
  }
}
```

Node.contains 首先调用 slot(of:) 方法，如果它找到了一个匹配的位置，那么我们所寻找的元素肯定在树中，所以 contains 应该返回 true。否则，我们可以使用返回的 index 值来将搜索范围缩小到某一个子节点中：

```
extension BTree.Node {
  func contains(_ element: Element) -> Bool {
    let slot = self.slot(of: element)
    if slot.match { return true }
    guard !children.isEmpty else { return false }
    return children[slot.index].contains(element)
  }
}
```

没错，B 树将有序数组算法和搜索树算法结合起来，创造了全新且让人激动的东西。(如果考虑 B 树的年代的话，称之为**全新**似乎有些夸张，不过它们的确让人心潮澎湃，不是吗？)

实现写时复制

实现写时复制的第一步，是使用我们到现在为止已经滚瓜烂熟的辅助方法来在必要的时候复制节点。对于根节点，没有什么特别可说的：

```
extension BTree {
  fileprivate mutating func makeRootUnique() -> Node {
    if isKnownUniquelyReferenced(&root) { return root }
    root = root.clone()
```

```
      return root
    }
}
```

打哈欠！

clone 方法需要对节点的属性进行浅复制。注意 elements 和 children 是数组，所以它们有它们自己的写时复制的实现。在这种情况下，这里有些冗余，但是它确实让我们的代码更短：

```
extension BTree.Node {
  func clone() -> BTree<Element>.Node {
    let clone = BTree<Element>.Node(order: order)
    clone.elements = self.elements
    clone.children = self.children
    return clone
  }
}
```

我们不需要复制变更计数器中的值，因为我们绝对不会在不同节点间比较这个值：这个值只在我们区分某一个节点实例的不同版本时会被使用。

子节点存在于一个数组中，而并不是独立的属性，所以它们的写时复制辅助方法稍有不同，我们需要添加一个参数来告诉函数我们之后想要改变的子节点到底是哪个：

```
extension BTree.Node {
  func makeChildUnique(at slot: Int) -> BTree<Element>.Node {
    guard !isKnownUniquelyReferenced(&children[slot]) else {
      return children[slot]
    }
    let clone = children[slot].clone()
    children[slot] = clone
    return clone
  }
}
```

幸运的是，Swift 的数组包含了一些非常秘密的实现方式，使得通过下标进行元素更改的操作是原地进行的。在这种实现方式下，我们可以对一个下标表达式进行

isKnownUniquelyReferenced 调用，而不会改变它的行为 (通常来说，我们只能对存储属性进行该调用。遗憾的是，在我们自己的集合类型中是无法实现这种**可变地址器** (mutating addressors) 的存取方式的；它背后的技术尚未成熟，而且只在标准库中可用。)

谓词工具 (Utility Predicates)

在继续之前，让我们先来做一点有意义的事，为节点值定义两个谓词属性：

```
extension BTree.Node {
  var isLeaf: Bool { return children.isEmpty }
  var isTooLarge: Bool { return elements.count >= order }
}
```

之后，我们将会使用 isLeaf 属性来判断一个实例是不是叶子节点；用 isTooLarge 来判断节点是否需要分割，若返回 true 则意味着节点拥有太多元素，需要被分割。

插入

是时候了，我们将按照引言中对本章的概述来实现插入操作，这里只需要将文字转换成实际的 Swift 代码即可。

我们从定义一个由单个元素和节点所组成的结构体开始，我把这个结构体称为一个**碎片** (splinter)。

```
extension BTree {
  struct Splinter {
    let separator: Element
    let node: Node
  }
}
```

碎片中的 separator 值一定要比它的 node 中的所有值都要小。正如其名，碎片就像是一个节点的小切片，它由一个单独的元素和跟随它的子节点组成。

接下来，定义一个方法来把某个节点分割为两部分，并将其中一部分提取为一个新的碎片：

```
extension BTree.Node {
  func split() -> BTree<Element>.Splinter {
    let count = self.elements.count
    let middle = count / 2

    let separator = self.elements[middle]

    let node = BTree<Element>.Node(order: order)
    node.elements.append(contentsOf: self.elements[middle + 1 ..< count])
    self.elements.removeSubrange(middle ..< count)

    if !isLeaf {
      node.children.append(contentsOf: self.children[middle + 1 ..< count + 1])
      self.children.removeSubrange(middle + 1 ..< count + 1)
    }
    return .init(separator: separator, node: node)
  }
}
```

这个函数中唯一值得注意的地方在于对索引的管理：稍不小心可能就会发生差一错误，从而破坏整棵树。

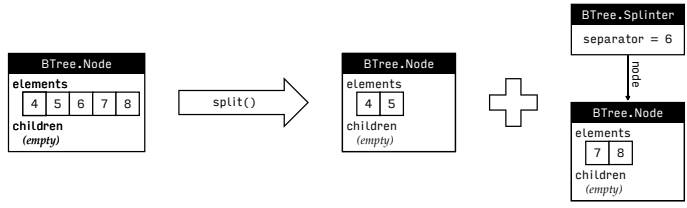


图 6.2: 将一个值为 **4--8** 的过大节点进行分割的结果。**split()** 操作返回的碎片中的节点包含了元素 **7** 和 **8**，原节点中的元素为 **4** 和 **5**。

接下来我们可以用碎片和分割方法将一个新元素插入到某个节点中了：

```
extension BTree.Node {
  func insert(_ element: Element) -> (old: Element?, splinter: BTree<Element>.Splinter?) {
```

这个方法首先会寻找节点中新元素所对应的位置。如果这个元素已经存在了，那么直接返回这个值，而不进行任何修改：

```
    let slot = self.slot(of: element)
    if slot.match {
      // 元素已经存在于树中。
      return (self.elements[slot.index], nil)
    }
```

否则，就需要实际进行插入，首先肯定需要将变更计数加一：

```
    mutationCount += 1
```

将一个新元素插入到叶子节点是很简单的：我们只需要将它插入到 elements 数组的正确的位置中就可以了。不过，这个额外加入的元素可能会使节点过大。当这种情况发生时，我们需要使用 split() 将节点分割为两半，并且将结果的碎片返回：

```
    if self.isLeaf {
      elements.insert(element, at: slot.index)
      return (nil, self.isTooLarge ? self.split() : nil)
    }
```

如果节点拥有子节点，那么我们需要通过递归调用来将它插入到子节点中正确的位置。insert 是一个可变方法，所以我们在需要的时候，我们应该调用 makeChildUnique(at:) 来创建写时复制的副本。如果递归的 insert 返回一个碎片，则需要把它插入到 self 中我们已经计算出的位置：

```
    let (old, splinter) = makeChildUnique(at: slot.index).insert(element)
    guard let s = splinter else { return (old, nil) }
    elements.insert(s.separator, at: slot.index)
    children.insert(s.node, at: slot.index + 1)
    return (nil, self.isTooLarge ? self.split() : nil)
```

```
  }
}
```

这样一来，当我们将一个元素插入到 B 树中一条全满路径的末端时，insert 将触发一系列的分割，最终将其从插入点向上一直传递到树的根节点。

如果路径上的所有节点都已经满了，那么最终根节点自身将被分割。这种时候，我们需要向树中添加新的层，具体的做法是：创建一个包含原根节点及得到的碎片的新根节点。实现这个处理的最佳位置是 BTree 结构体的公有 insert 方法之中，当然了，我们还必须牢记，对树进行任何变更之前务必先调用 makeRootUnique 方法：

```
extension BTree {
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪  Element) {
    let root = makeRootUnique()
    let (old, splinter) = root.insert(element)
    if let splinter = splinter {
      let r = Node(order: root.order)
      r.elements = [splinter.separator]
      r.children = [root, splinter.node]
      self.root = r
    }
    return (old == nil, old ?? element)
  }
}
```

这就是将元素插入到 B 树中所需要做的全部了。坦诚地讲，这些代码不如 RedBlackTree 中的优雅，但是对比我们在 RedblackTree2 所写的，已经好太多了。

对插入进行性能测试，可以得到 图 6.3 中的结果。对于小集合，显然我们的性能和 SortedArray 在同等范围中：BTree.insert 仅仅只慢了 10-15%。好的地方在于，对于大的数据集，同样的代码对比 RedblackTree2 的插入速度快了有 3.5 倍！通过随机插入 400 万个元素的方式创建一棵 BTree 仅仅只需要三秒。

作为一棵平衡搜索树，B-tree.insert 和 RedBlackTree 以及 RedblackTree2 一样，拥有 $O(\log n)$ 的渐进复杂度，但是它借鉴并汲取了 SortedArray 的内存访问模式，这使得对于不论

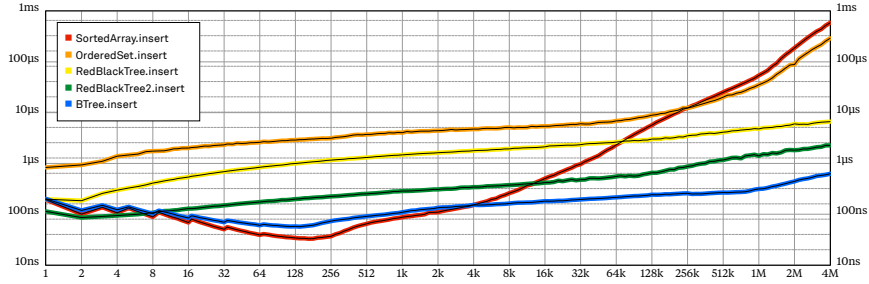


图 6.3: 对比 **SortedSet.insert** 的五种不同实现的性能。

大小的任意输入尺寸，B 树的性能都能得到令人欣喜的提升。有时候组合两种数据结构会给我们带来 1 + 1 > 2 的效果。

实现集合类型

实现 Collection 的时候，所面临的最大设计挑战不外乎于如何选定一个优秀的索引类型。我们将为 B 树选择一条和 RedblackTree2 相仿的路，让每个索引包含树中的一条完整路径。

B 树路径

B 树中的一条路径可以由从根节点开始的一系列位置构成。不过为了更高效一些，我们也会将路径上对相应节点的引用包含进来。

索引不能包括对集合类型的强引用。在 RedblackTree2 中，我们一开始使用了弱引用来满足这个要求。这次，我们将利用上一章最后的结论，直接使用 unsafe unowned 引用。我们已经知道，这样的引用不会带来引用计数的额外开销，所以它们会稍微快一些。这种小改动在迭代测试中的累加效应相当明显，对于 RedblackTree2 来说，它为迭代带来了 200% 的加速。

作为拥有卓越原生性能的交换，语言层面对于 unowned(unsafe) 的去引用操作完全没有提供安全性保障。它们的引用目标可能已不再存在，包含的数据也许并不在预期之内。从这个角度来说，这类引用的工作方式和 C 指针可以说是彼此彼此。

对于有效的索引来说，这没有任何问题，因为它们所引用的树和它们被创建时的树的状态是一致的。但是对于无效的索引就必须特别小心了，因为它们的路径上的节点引用可能已经被破坏

了。访问已经破坏的引用会造成不可预期的结果，app 可能会崩溃，或者默默给你返回不正确的数据。(这和简单的 unowned 引用稍有不同，unowned 还是做了一些引用计数的工作，来确保引用目标消失时你的程序会发生崩溃。)

说了这么多，让我们开始写代码吧。我们将使用一个 UnsafePathElement 数组来表示一条 B 树路径。UnsafePathElement 结构体定义如下，它包含一个对节点的引用以及一个表示位置的整数：

```
extension BTree {
  struct UnsafePathElement {
    unowned(unsafe) let node: Node
    var slot: Int

    init(_ node: Node, _ slot: Int) {
      self.node = node
      self.slot = slot
    }
  }
}
```

上述定义中的存储属性 node 的声明看起来有点可怕。

我们还会定义一系列计算属性，用来访问路径元素的一些基本特性，比如路径元素所引用的值，这个值之前的对应的子节点，元素上的节点是不是一个叶子节点，以及路径元素是否指向节点的末尾等：

```
extension BTree.UnsafePathElement {
  var value: Element? {
    guard slot < node.elements.count else { return nil }
    return node.elements[slot]
  }
  var child: BTree<Element>.Node {
    return node.children[slot]
  }
  var isLeaf: Bool { return node.isLeaf }
  var isAtEnd: Bool { return slot == node.elements.count }
}
```

注意，路径元素的位置有可能恰好在节点内最后一个元素之后，这种情况下这个路径元素将没有对应的 value 值。不过对于中间节点来说，其中的每个位置都会对应存在一个子节点。

如果还能比较路径元素的相等性就再好不过了，所以让我们来实现 Equatable：

```
extension BTree.UnsafePathElement: Equatable {
  static func ==(left: BTree<Element>.UnsafePathElement, right:
  ↪  BTree<Element>.UnsafePathElement) -> Bool {
    return left.node === right.node && left.slot == right.slot
  }
}
```

B 树索引

下面是 BTreeIndex 的定义。它看起来和 RedblackTree2Index 很相似，在索引中它将持有一个路径元素的序列，对根节点的弱引用，以及索引被创建时的变更次数。注意，对于根节点我们还是使用了弱引用，这是因为弱引用将允许我们在没有具体的树的值和索引中的值进行匹配的时候，可以使用索引自身来进行验证：

```
extension BTree {
  public struct Index {
    fileprivate weak var root: Node?
    fileprivate let mutationCount: Int64

    fileprivate var path: [UnsafePathElement]
    fileprivate var current: UnsafePathElement
```

大多数情况下，在 B 树中可以简单地通过增加最终路径元素的位置值，来进行索引步进。将这个 “热点” 元素从数组里拿出来，单独存储在一个 current 属性里，可以让这种通用处理稍微快一些。(Array 自身的索引验证，以及其固有的对数组底层存储缓冲区进行的非直接访问，将会带来些许的开销增加。) 这样的微小优化一般来说是不必要的 (或者甚至是有害的)。但是，我们已经决定了使用不安全的引用，相比起来，这就显得微不足道了，况且毫无疑问这几乎没什么危害。

我们还需要两个内部初始化方法，来创建树的 startIndex 和 endIndex。前者将构造一条通向树中首个元素的路径，而后者只用将路径设在在根节点中最后一个元素之后即可：

```
    init(startOf tree: BTree) {
      self.root = tree.root
      self.mutationCount = tree.root.mutationCount
      self.path = []
      self.current = UnsafePathElement(tree.root, 0)
      while !current.isLeaf { push(0) }
    }

    init(endOf tree: BTree) {
      self.root = tree.root
      self.mutationCount = tree.root.mutationCount
      self.path = []
      self.current = UnsafePathElement(tree.root, tree.root.elements.count)
    }
  }
}
```

理论上说，startIndex 的复杂度是 $O(\log n)$。不过我们已经看到，实际上 B 树的深度更接近于 $O(1)$，所以在这种情况下，我们完全没有违背 Colleciton 中关于性能的要求。

索引验证

因为空的 B 树也有一个根节点，所以在任意有效的 B 树索引中根节点的引用不能是 nil。除此以外，B 树的索引验证的方式基本上和 RedblackTree2 中的一样。

当想要变更 BTree 的值时，需要先仔细考虑要么改变根节点引用，要么改变根节点变更计数。这样一来，当某个索引中根节点引用和变更计数两者都匹配时，我们就知道该索引依然有效，也就能安全地使用它的路径上的元素了：

```
extension BTree.Index {
  fileprivate func validate(for root: BTree<Element>.Node) {
    precondition(self.root === root)
    precondition(self.mutationCount == root.mutationCount)
  }

  fileprivate static func validate(_ left: BTree<Element>.Index, _ right:
  ↪  BTree<Element>.Index) {
```

```
      precondition(left.root === right.root)
      precondition(left.mutationCount == right.mutationCount)
      precondition(left.root != nil)
      precondition(left.mutationCount == left.root!.mutationCount)
    }
}
```

我们将会在 Equatable 和 Comparable 的实现中用上 static 版本的索引验证方法。由于这个方法的存在，我们不能将对根节点的弱引用转换为 unowned(unsafe)，因为我们需要在不从外部提供树的参照的情况下，也能完成索引的验证工作。

索引导航

要在树中导航，我们需要定义两个辅助方法：push 和 pop。push 接受一个与当前路径相关的子节点中的位置值，并把它添加到路径的末端。pop 则负责将路径的最后一个元素移除：

```
extension BTree.Index {
  fileprivate mutating func push(_ slot: Int) {
    path.append(current)
    let child = current.node.children[current.slot]
    current = BTree<Element>.UnsafePathElement(child, slot)
  }

  fileprivate mutating func pop() {
    current = self.path.removeLast()
  }
}
```

有了这两个函数，我们就能定义在树中从一个索引步进到下一个元素的方法了。对于绝大多数情况来说，这个方法要做的仅仅是增加当前路径元素 (也就是最后一个元素，current) 的位置值。仅有的例外发生于 (1) 对应的叶子节点中没有更多的元素时，或者 (2) 当前节点不是一个叶子节点时。(两种情况相对来说都是很罕见的。)

```
extension BTree.Index {
  fileprivate mutating func formSuccessor() {
```

```
    precondition(!current.isAtEnd, "Cannot advance beyond endIndex")
    current.slot += 1
    if current.isLeaf {
      // 这个循环很可能一次都不会执行。
      while current.isAtEnd, current.node !== root {
        // 上溯到最近的，拥有更多元素的祖先节点。
        pop()
      }
    }
    else {
      // 下行到当前节点最左侧叶子节点的开头。
      while !current.isLeaf {
        push(0)
      }
    }
  }
}
```

寻找前置索引与此相似，我们需要稍微重新组织一下代码，因为我们想要找的是节点起始位置，而非结尾：

```
extension BTree.Index {
  fileprivate mutating func formPredecessor() {
    if current.isLeaf {
      while current.slot == 0, current.node !== root {
        pop()
      }
      precondition(current.slot > 0, "Cannot go below startIndex")
      current.slot -= 1
    }
    else {
      while !current.isLeaf {
        let c = current.child
        push(c.isLeaf ? c.elements.count - 1 : c.elements.count)
      }
    }
  }
}
```

上面的函数都是私有的辅助方法，所以完全可以假设当它们被调用时，索引已经被它们的调用者验证过了。

比较索引

索引需要满足 Comparable，所以我们还需要实现索引的判等和小于等于操作符。这些函数是公有入口，因此我们必须记住在访问它们路径上的任意元素之前，先对索引进行验证：

```
extension BTree.Index: Comparable {
  public static func ==(left: BTree<Element>.Index, right: BTree<Element>.Index) -> Bool {
    BTree<Element>.Index.validate(left, right)
    return left.current == right.current
  }

  public static func <(left: BTree<Element>.Index, right: BTree<Element>.Index) -> Bool {
    BTree<Element>.Index.validate(left, right)
    switch (left.current.value, right.current.value) {
    case let (a?, b?): return a < b
    case (nil, _): return false
    default: return true
    }
  }
}
```

实现 Collection

现在我们已经具备了让 BTree 满足 BidirectionalCollection 的所有条件了；在每个方法的实现中，我们只需要调用上述索引方法即可完成具体工作。此外，还需要确保在调用前已经对索引进行了验证，因为我们并没有在索引上实现验证其自身的功能：

```
extension BTree: SortedSet {
  public var startIndex: Index { return Index(startOf: self) }
  public var endIndex: Index { return Index(endOf: self) }

  public subscript(index: Index) -> Element {
```

```
    index.validate(for: root)
    return index.current.value!
  }

  public func formIndex(after i: inout Index) {
    i.validate(for: root)
    i.formSuccessor()
  }

  public func formIndex(before i: inout Index) {
    i.validate(for: root)
    i.formPredecessor()
  }

  public func index(after i: Index) -> Index {
    i.validate(for: root)
    var i = i
    i.formSuccessor()
    return i
  }

  public func index(before i: Index) -> Index {
    i.validate(for: root)
    var i = i
    i.formPredecessor()
    return i
  }
}
```

获取元素个数

使用索引进行迭代现在只消耗 $O(n)$ 的时间，但是所有那些索引验证的工作将让操作变慢很多。迫于此，我们可以考虑为像是 count 属性这样的基本的 Collection 成员，准备一些特殊的实现：

```
extension BTree {
  public var count: Int {
    return root.count
```

```
    }
  }

  extension BTree.Node {
    var count: Int {
      return children.reduce(elements.count) { $0 + $1.count }
    }
  }
```

注意，Node.count 在 reduce 的闭包里使用了递归调用，这需要访问到 B 树中的每个节点，由于技术上来说我们会有 $O(n)$ 个节点，所以这个计数操作也将在 $O(n)$ 时间下完成。(虽然这已经比一个元素一个元素计数要快得多了。)

> 由于 B 树的节点一般都很大，所以在每个节点中包含一个子树的元素个数值是一个不错的想法，这会使数据结构变为一棵**顺序统计树**。这么做可以将 count 变为一个 $O(1)$ 操作，并且让我们在 $O(\log n)$ 时间内就能查找到树中的第 *i* 个元素。我们这里就不给出实现了，你可以在我们官方的 BTree 项目里找到对于这个想法的完整实现。

自定义迭代器

我们还需要自定义一个迭代器类型，这样我们才能将索引验证的开销从简单的 for-in 循环中移除。下面是一种直接的实现，我们在 RedblackTree2 中已经用过同样的方式了：

```
extension BTree {
  public struct Iterator: IteratorProtocol {
    let tree: BTree
    var index: Index

    init(_ tree: BTree) {
      self.tree = tree
      self.index = tree.startIndex
    }

    public mutating func next() -> Element? {
      guard let result = index.current.value else { return nil }
```

```
        index.formSuccessor()
        return result
      }
    }

    public func makeIterator() -> Iterator {
      return Iterator(self)
    }
}
```

迭代性能

为了优化 B 树中的迭代性能，我们用上了十八般武艺：

- → **算法改进：**索引是由树中的完整路径来表示的，所以索引的步进平摊下来的时间复杂度是 $O(1)$。
- → **为一般情形进行优化：**B 树的绝大部分值都存储在叶子节点的 elements 数组中。通过将最后一个路径元素提出来放在自己的存储属性中，我们可以确保在这样的元素间步进时所需要的操作尽可能少。
- → **提升局部访问性**：通过将我们的值按照缓存的尺寸排列为合适大小的数组，我们可以构建出完美适合我们计算机多层内存架构的数据结构。
- → **使用捷径完成验证：**我们实现了自定义的迭代器，这样我们就可以在迭代的时候将重复的索引验证去除掉。
- → **谨慎使用不安全的操作：**在路径内部，我们使用了不安全的引用来指向树的节点，这样，创建或修改索引路径就不会再涉及引用计数操作了。索引验证为我们保驾护航，因此我们绝不会遇到被破坏的引用。

那么，B 树中的迭代到底能有多快呢？快的离谱！图 6.4 是我们的测试结果。

我们已经很满意 RedblackTree2 最终的迭代性能了，不过它和 BTree 完全不在一个档次：BTree 要快上 40 倍！不过相比起来，SortedArray 还是比 BTree 快 8 倍，所以我们还有一些改进的空间。

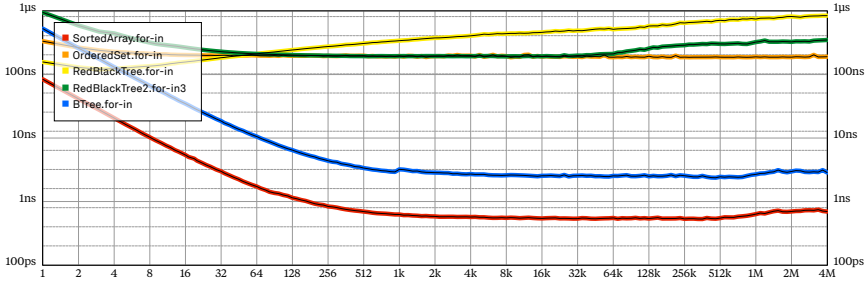


图 **6.4:** 对比五种 **SortedSet** 实现的迭代性能。

例子

```
var set = BTree<Int>(order: 5)
for i in (1 ... 250).shuffled() {
  set.insert(i)
}
▶ set
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
↪  23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
↪  42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
↪  61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
↪  80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
↪  99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113,
↪  114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
↪  129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
↪  144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158,
↪  159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173,
↪  174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188,
↪  189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203,
↪  204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218,
↪  219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233,
↪  234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248,
↪  249, 250]
```

```
let evenMembers = set.reversed().lazy.filter { $0 % 2 == 0 }.map { "\($0)"
↪   }.joined(separator: ", ")
▶ evenMembers
  250, 248, 246, 244, 242, 240, 238, 236, 234, 232, 230, 228, 226, 224, 222,
↪   220, 218, 216, 214, 212, 210, 208, 206, 204, 202, 200, 198, 196, 194, 192,
↪   190, 188, 186, 184, 182, 180, 178, 176, 174, 172, 170, 168, 166, 164, 162,
↪   160, 158, 156, 154, 152, 150, 148, 146, 144, 142, 140, 138, 136, 134, 132,
↪   130, 128, 126, 124, 122, 120, 118, 116, 114, 112, 110, 108, 106, 104, 102,
↪   100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68, 66, 64,
↪   62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26,
↪   24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2
```

性能汇总

图 6.5 汇总了 BTree 在我们的四种标准微性能测试中的表现。值得指出的是，使用 for-in 的迭代现在要比用 forEach 的版本快得多，这和我们在红黑树中遇到的情况完全相反。

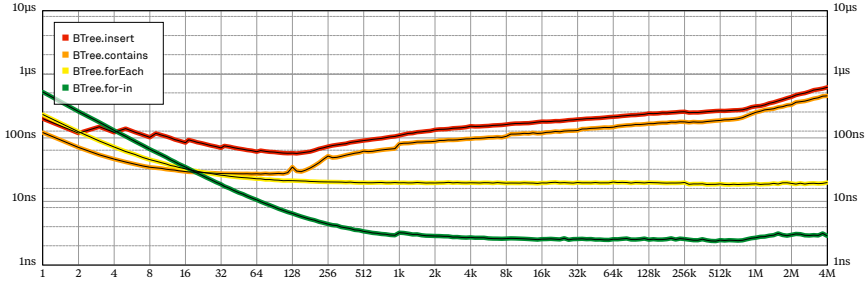


图 6.5: BTree 操作的性能测试结果。

我们使用 Int 作为元素，并在一台 64 位，拥有 32KB 的 L1 缓存的 MacBook 上进行性能测试，这样一来，BTree<Int> 的默认阶数为 1,024。所以，第一次节点分割将发生在我们插入第 1,024 个元素的时候，分割后树将从一个单一节点转换成排列为两层的三个节点。在图中元素数量达到 1,000 的地方，我们可以清晰地看到曲线突然发生了一个向上的跳跃。

树扩展为三层时的尺寸是无法精确预言的，它将发生在五十万到一百万个元素之间。insert 和 contains 的曲线在这个区间内表现出更快速的增长，这和多加入一层的变化是一致的。

为了让 B 树达到四层，我们需要插入大约十亿个整数。图 6.6 将 BTree.insert 的图表扩展到了这样一个怪兽级的大数据集。比较明显的是，插入曲线分为三个阶段，分别对应了 B 树中一层，两层和三层的情况。在图表最右侧，我们可以清晰地看到曲线的上翘，这表明了出现了第四层树。不过在此时，性能测试已经将 MacBook 的 16 GB 物理内存消耗殆尽，这使得 macOS 开始进行内存页面压缩，它甚至会将一些数据转存到 SSD 上。显然，最后的突然增长有一些 (或者说大部分？) 是由于页面操作，而非树的第四层带来的影响。对于测试用的这台机器来说，想将性能测试再往右延伸，会超出其本身的限制，是不可能的了。

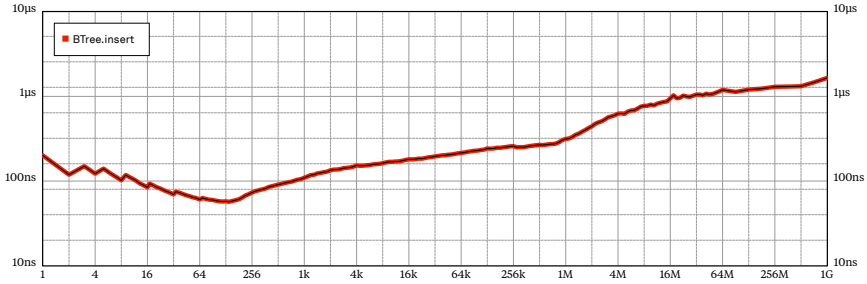


图 6.6: 通过随机插入创建 **B** 树。

额外优化

7

在本章中，我们将集中讨论一下如何进一步优化 BTree.insert，并努力将代码最后的潜力挖掘出来。

我们会创建另外三种 SortedSet 的实现，并 “开创性” 地将它们命名为 BTree2，BTree3 和 BTree4。为了让本书保持在合理的长度，我们将不会把这三个版本的 B 树的完整代码写出来，只会通过一些具有代表性的代码片段来描述发生的改变。如果你想要参考所有三个版本的 B 树的完整源码，可以去看一看本书的 GitHub 仓库。

如果你已经对 SortedSet 感到厌倦了，跳过本章也没问题。因为这里描述的一些进阶技巧在日常的 app 开发中很少会被用到。

内联 Array 方法

BTree 将元素和 Node 的子节点存储在标准的 Array 中。在上一章里，这使得代码相对容易理解，也对我们认识 B 树起到了帮助。然而，Array 中包含了索引验证和写时复制的逻辑，这和 BTree 中的相关逻辑重复了。如果我们的代码没有任何问题，那么 BTree 将永远不会使用越界的下标访问数组，而且 B 树自己也实现了写时复制的行为。

看看我们的插入性能测试图表 (图 7.1)，可以看到，当集合尺寸相对较小时，虽然 BTree.insert 已经十分接近 SortedArray.insert 了，但它们之间仍然有 10%-20% 的性能差距。消除 Array 的 (微小) 开销是否足以填补这个性能差距？让我们试试看吧！

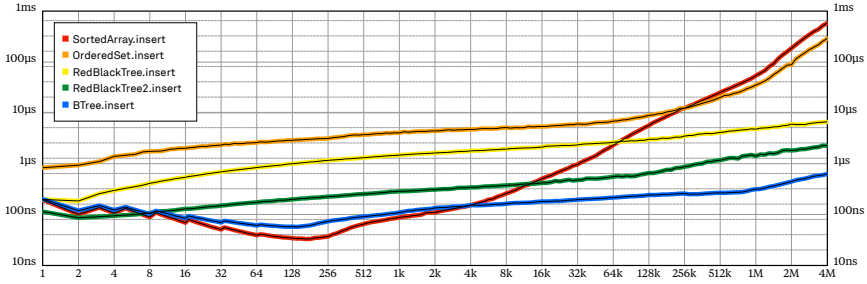


图 7.1: 对比五种不同 **SortedSet.insert** 实现的性能。

Swift 标准库中包含了 UnsafeMutablePointer 和 UnsafeMutableBufferPointer 类型，我们可以用它们来实现我们自己的存储缓冲区。它们的名字有些可怕，但却名符其实。和这些类型打

交道只比和 C 指针打交道好那么一点点，代码上差之毫厘，往往导致结果谬以千里，稍有不慎可能就会造成难以调试的内存污染，内存泄漏，甚至是更糟糕的问题。换个角度来看，如果我们能够细心谨慎地使用这些类型，也许可以通过使用这些类型让我们的性能得到稍许提升。

那么让我们开始写 BTree2 吧，这是我们第二次尝试实现 B 树。作为开始的是一些人畜无害的代码：

```
public struct BTree2<Element: Comparable> {
  fileprivate var root: Node

  public init(order: Int) {
    self.root = Node(order: order)
  }
}
```

不过这次，在 Node 中，我们将 elements 数组转换为了一个不安全的可变指针，这个指针指向一个手动申请的缓冲区的起始位置。因为指针并不会自己去追踪计数，所以我们还需要添加一个存储属性，用来表示当前缓冲区内的元素个数：

```
extension BTree2 {
  class Node {
    let order: Int
    var mutationCount: Int64 = 0
    var elementCount: Int = 0
    let elements: UnsafeMutablePointer<Element>
    var children: ContiguousArray<Node> = []
```

在 BTree2 中，我们不打算动 children，它将保持依然是一个数组。虽然我们将 children 的类型从 Array 改成了有时候会稍微快一些的 ContiguousArray，但本质上它依然是数组。因为绝大多数元素都存在于叶子节点中，所以加速中间节点很可能无法得到站在全局可以观察到的改进，我们最好不要在优化中间节点上花太大力气。

Node 的指定初始化方法 (designated initializer) 负责为我们的元素缓冲区申请内存。我们会申请能存放下 order 个数元素的空间，这样，缓冲区能够持有的元素个数将会比我们允许的最大尺寸多一。这非常重要，如此一来，在我们对节点进行分割之前，就可以让节点暂时超出最大个数这一限制：

```
    init(order: Int) {
      self.order = order
      self.elements = .allocate(capacity: order)
    }
```

在前面的章节中，我们没有使用 Array.reserveCapacity(_:) 来预先为我们的两个数组申请存储空间，而是依赖了 Array 的自动存储管理。这会让代码简单一些，但同时也导致了两个不太理想的结果。第一，当我们将一个新的元素插入到 BTree 节点中时，有时候 Array 会需要申请一个全新且更大的缓冲区，并将已经存在的元素移动到新的缓冲区里去。这会给插入操作带来一些额外的开销。第二，Array 在扩大存储缓冲区时，是以 2 的幂次关系递增的，也就是说，有可能树的阶仅仅只是 “稍微” 比某个值大了一点，但 Array 申请的空间却了比整份代码所需要的多出了 50%。可以通过申请正好和结点的最大尺寸相同的缓冲区，来同时避免这两个问题。

因为我们是手动申请的缓冲区，自然还需要在某个时间点手动释放它们。自定义的 deinit 方法是进行这个操作最好的地方：

```
    deinit {
      elements.deinitialize(count: elementCount)
      elements.deallocate(capacity: order)
    }
  }
}
```

注意，我们必须对元素占用的内存进行明确的逆初始化 (deinitialize)，即，在回收内存之前将缓冲区恢复到最初的未初始化状态。这可以保证在 Element 值中可能存在的引用被正确释放，甚至可能让那些仅由缓冲区持有的值被正确回收。要是没有逆初始化这些值，有可能会导致内存泄漏：

对节点的复制操作可以很好地诠释 elements 缓冲区的用法。我们调用 initialize(from:count:) 方法来将数据在缓冲区之间进行复制，它会在需要的时候负责更新引用计数。如果节点是一个内部节点，我们也可以通过在新的节点的 children 数组上调用 reserveCapacity(_:) 来预先申请足够的空间，以容纳我们可能会需要的子节点。如果不是内部节点，我们将会保持 children 为空，这样就不会因为永远不会用到的存储而浪费内存空间了：

```
extension BTree2.Node {
  func clone() -> BTree2<Element>.Node {
```

```
    let node = BTree2<Element>.Node(order: order)
    node.elementCount = self.elementCount
    node.elements.initialize(from: self.elements, count: self.elementCount)
    if !isLeaf {
      node.children.reserveCapacity(order + 1)
      node.children += self.children
    }
    return node
  }
}
```

在插入中我们所需要的 split() 操作也能受益于 UnsafeMutablePointer 对移动初始化 (move initialization) 的支持。在原来的代码里，为了移动元素，我们需要两个步骤：首先将它们复制到新的数组中，然后再将它们从原来的数组中删除。当 Element 包含带有引用计数的值的时候，这种合二为一的移动操作会快得多。(不过对于像是 Int 这样的简单值类型来说，产生的性能影响是微乎其微的。)

```
extension BTree2.Node {
  func split() -> BTree2<Element>.Splinter {
    let count = self.elementCount
    let middle = count / 2

    let separator = elements[middle]
    let node = BTree2<Element>.Node(order: self.order)

    let c = count - middle - 1
    node.elements.moveInitialize(from: self.elements + middle + 1, count: c)
    node.elementCount = c
    self.elementCount = middle

    if !isLeaf {
      node.children.reserveCapacity(self.order + 1)
      node.children += self.children[middle + 1 ... count]
      self.children.removeSubrange(middle + 1 ... count)
    }
    return .init(separator: separator, node: node)
  }
}
```

要将一个新元素插入到缓冲区的中间，我们需要实现与 Array.insert 等效的方法。要做到这一点，我们首先要从插入点开始，将已有的元素向右移动一个位置，为新元素腾出空间：

```
extension BTree2.Node {
  fileprivate func _insertElement(_ element: Element, at slot: Int) {
    assert(slot >= 0 && slot <= elementCount)
    (elements + slot + 1).moveInitialize(from: elements + slot, count: elementCount -
    ↪ slot)
    (elements + slot).initialize(to: element)
    elementCount += 1
  }
}
```

为了让原来的 insert 代码在 BTree2 中也可以使用，我们需要做一些适配工作：

```
extension BTree2.Node {
  func insert(_ element: Element) -> (old: Element?, splinter: BTree2<Element>.Splinter?)
  ↪ {
    let slot = self.slot(of: element)
    if slot.match {
      // 元素已经在树中
      return (self.elements[slot.index], nil)
    }
    mutationCount += 1
    if self.isLeaf {
      _insertElement(element, at: slot.index)
      return (nil, self.isTooLarge ? self.split() : nil)
    }
    let (old, splinter) = makeChildUnique(at: slot.index).insert(element)
    guard let s = splinter else { return (old, nil) }
    _insertElement(s.separator, at: slot.index)
    self.children.insert(s.node, at: slot.index + 1)
    return (old, self.isTooLarge ? self.split() : nil)
  }
}

extension BTree2 {
```

```
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪   Element) {
    let root = makeRootUnique()
    let (old, splinter) = root.insert(element)
    if let s = splinter {
      let root = BTree2<Element>.Node(order: root.order)
      root.elementCount = 1
      root.elements.initialize(to: s.separator)
      root.children = [self.root, s.node]
      self.root = root
    }
    return (inserted: old == nil, memberAfterInsert: old ?? element)
  }
}
```

要创建一个 BTree2，原则上我们要将 Array 的实现从标准库中拿出来，移除掉不需要的功能，并把它内嵌到我们的 B 树代码中。

性能测试的结果如图所示 图 7.2。插入操作的速度稳定地提升了 10-20%。我们填补上了 B 树和 SortedArray 之间最后的性能差距：至此，BTree2.insert 在全范围内的性能持平或是超越了所有之前的 SortedSet 实现。

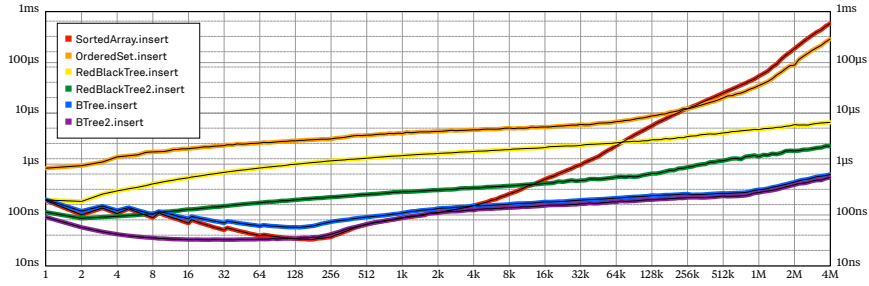


图 7.2: 对比六种 **SortedSet** 实现的插入性能。

作为额外收益，移除 Array 的索引验证检查，让迭代的性能也提高了两倍；参见 图 7.3. BTree2.for-in 现在只比 SortedArray 慢四倍了；这是一个很显著的进步！

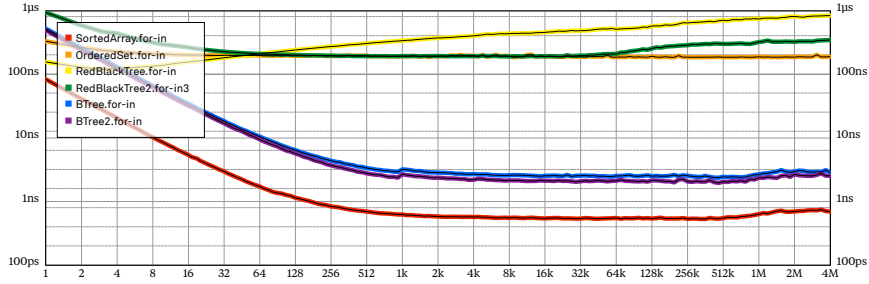


图 7.3: 对比六种 **SortedSet** 实现的迭代性能。

优化对共享存储的插入

到目前为止，每当测量插入性能时，我们总是假设我们的有序集合的存储是完全独立的。但是我们从没有测试过如果我们将元素插入使用共享存储的集合中时，会发生什么。

为了了解将数据插入共享存储的性能，让我们来设计一种新的性能测量方式吧！一种办法是在每次插入元素后都对整个集合进行复制，然后测量插入一系列元素所花费的时间：

```
extension SortedSet {
  func sharedInsertBenchmark(_ input: [Element]) {
    var set = Self()
    var copy = set
    for value in input {
      set.insert(value)
      copy = set
    }
    _ = copy // 避免变量没有被读取的警告。
  }
}
```

图 7.4 展示了对我们到目前为止所实现的 SortedSet 进行这一新的性能测试所得到的结果。

显然，我们对 NSOrderedSet 的封装并不是为了针对这种滥用的情况而存在的。一旦元素数量达到几千，它的性能就会比 SortedArray 慢大约一千倍。不过 SortedArray 也没有好多少：为了将存储和复制分离，这两种基于数组的有序集合都需要对存储于其中的每一个值进行完全复

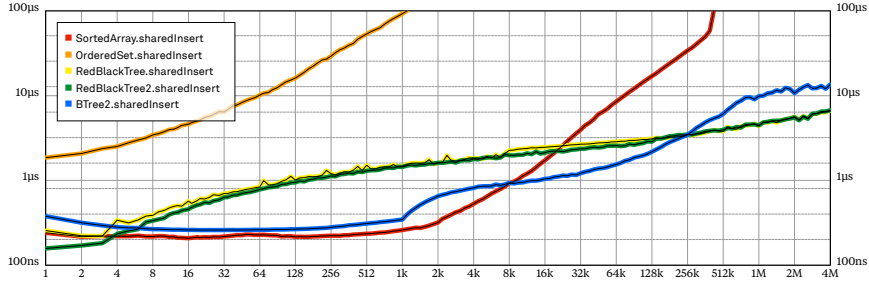


图 7.4: 向共享存储中进行一次插入操作的平摊时间。

制。这不会改变它们的插入操作的渐进性能 (依然是 $O(n)$)，但却会为其附加上一个相当可观的常数系数。对 SortedArray 来说，sharedInsert 性能测试要比普通的 insert 慢大约 3.5 倍。

两种红黑树的实现表现得好很多：对于每一次插入操作，它们只需要对落在新插入元素的路径上的节点进行复制。RedBlackTree 不论何时都会这么做：它的插入性能只和节点是否被共享相关。但是 RedBlackTree2 无法使用原地变更：所有的 isKnownUniquelyReferenced 的调用都会返回 false，所以在这个特别的性能测试中，它比 RedBlackTree 要稍微慢一些。

BTree2 最初性能相当好，但是在大约 64,000 个元素时，它突然就变慢了。在这个阶段，树即将增长到三层，它的 (第二层) 根节点中包含了太多的元素，以致于创建一份复制的耗时简直可以和插入操作相提并论。随着树变为三层，这个情况将愈发严重，最后它的性能要比红黑树慢 6 倍左右。(BTree.insert 的平摊性能保持在 $O(\log n)$，变慢只是因为添加了一个巨大的常数系数。)

我们希望我们的 B 树在所有图里的速度都遥遥领先于红黑树。那我们能做些什么来防止在共享存储的情况下的这种性能衰退吗？问得好，我们当然有办法！

我们推测，变慢是由于大量的内部节点所导致的。我们也知道，树中绝大部分的值都是存储在叶子节点中的，所以内部节点**通常**来说并不会对 B 树性能造成很大影响。在这个前提下，我们可以按照我们的想法来任意改造内部节点，而不必担心它会对性能图表产生什么巨大影响。那么，如果我们大幅限制中间节点的最大尺寸，同时保持叶子节点的尺寸不变，会怎么样呢？

要实现这个其实非常简单，我们只需要在 BTree2.insert 中进行一行很小的改动即可：

```
extension BTree3 {
  @discardableResult
```

```
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪   Element) {
    let root = makeRootUnique()
    let (old, splinter) = root.insert(element)
    if let s = splinter {
      let root = BTree3<Element>.Node(order: 16) // <--
      root.elementCount = 1
      root.elements.initialize(to: s.separator)
      root.children = [self.root, s.node]
      self.root = root
    }
    return (inserted: old == nil, memberAfterInsert: old ?? element)
  }
}
```

代码块中被标记出来的一行，为树添加了一个新层。我们用来作为新的根节点的阶的数字，也会被用作所有对它进行分割后所得到的节点的阶数。在这里，通过使用一个小的阶数来代替 self.order，我们确保了**所有的**内部节点都以这个阶数进行初始化，而不是以原来初始化 BTree 时所用的阶数。(新的叶子节点总是由已存在的叶子节点分割而成，所以这个值不会应用于叶子节点。)

我们将这个新版本的 B 树命名为 BTree3，运行性能测试，可以得到 图 7.5 中的结果。上面的推测是正确的；通过限制内部节点的尺寸，性能得到了很大提升！BTree3 现在即使在大数据集的情况下，也比 RedBlackTree 快上 2-2.5 倍。(通过这种费力的方式创建一棵含有四百万个元素的 BTree3 只需要 15 秒；而 BTree2 做同样的事要花 10 倍的时间。)

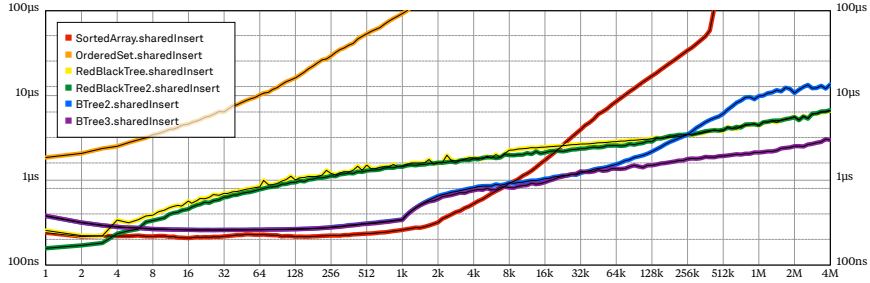


图 7.5: 向共享存储中进行单次插入的平摊时间。

限制内部节点的尺寸通常会增加树的高度，这确实会影响到一部分 B 树的操作。不过，大部分的影响都是可以忽略的：它只会使 contains 和原地的 insert 变慢约 10%，而且它对迭代方法完全没有影响。比如，图 7.6 比较了 BTree3 和我们的其他实现的原地插入操作的性能。

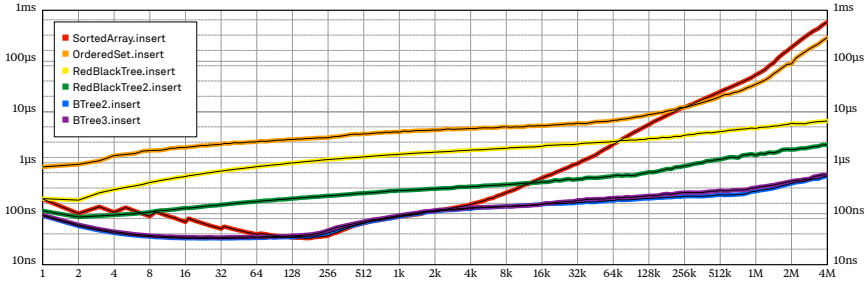


图 7.6: 向共享存储中进行单次插入的平摊时间。

比想象的要简单不少，对吧?

移除冗余复制

在实现写时复制的时候，我们的代码无一例外地使用了 makeFooUnique 这类方法，以在必要时创建共享存储的复制。之后，我们的代码再分别修改上一步创建的独立存储。

如果你仔细想想，会发现这么做是一种相对不那么有效率的方法：比如，当我们需要在一个共享的 OrderedSet 的开头插入新元素的时候，makeUnique 首先会用完全相同的元素创建一个全新的 NSOrderedSet，然后 insert 立即进行处理，它会把所有的值向右移动一个位置，给新的元素挪出空间。

所以 OrderedSet 的 sharedInsert 测试这么慢就一点都不奇怪了！如果我们在创建共享存储的复制时候已经在正确的位置插入了新元素的话，性能就会得到一定程度的提升。不过我们不会为 OrderedSet 实现这个改进，因为能带来的提升并不多，就算这么做了，与其他的 SortedSet 的实现相比，它的性能还是要差很多。

不过，我们也能从至今为止我们所实现的所有三种 BTree 中看到同样的非最优的行为。当我们向一个共享的 B 树节点插入元素的时候，我们首先会将所有存在的元素复制到新的缓冲区；接下来在一个单独的步骤中插入新的值。更糟糕的是，插入操作还有可能导致节点过大，这时还

需要进行分割，将一半的元素移动到第三个缓冲区去，而这件事只能在另一次单独的操作中完成。这么一来，单次的插入操作中，对数百个元素进行复制或移动操作的次数可能不是一次，也不是两次，而是**多达三次**！这相当低效。

可以通过将 makeUnique、insert 和 split 合并到一个单独的，相对复杂的操作中，从而把所有不必要的复制和移动操作去掉。要将 makeUnique 和 insert 做合并，我们可以实现一个不可变的插入操作，然后，若侦测到节点是共享的，则切换到不可变版本。要将这个混合的插入操作与 split 统一起来，我们需要在插入**之前**就检测是否需要分割操作。如果需要分割，那么就从头创建两个节点，这样新元素就已经位于正确的位置了。注意，这个新的元素可能会位于第一个节点里，也可能位于第二个节点里，或者甚至正好落在中间，这样的话它就将成为原来节点的两半的分隔值。因此，要将这三种操作统一起来，我们需要写 $2 \times 4 = 8$ 种独立的插入操作。

我尽自己所能实现了这些方法，但是不幸的是，得到的结果有出有入。共享的插入操作对于特定的元素数量变快了一些，但对于其它的元素数量却又变慢了一些。原地的插入操作没有太大的改变。

这里我不会将我的 BTree4 的代码列在这里，但是你可以在 GitHub 仓库中找到它们。不妨试着调整一下它，也许你可以使它变得更快！

总结

8

在本书中，针对同一个简单的集合类型，我们已经讨论了七种不同的实现方法。每次我们创建一种新的解决方案，我们的代码就变得比之前复杂一些，但是作为交换，我们获取了可观的性能提升。也许用性能测试图表中随着插入实现的进展而稳步下降的曲线最能说明这一点。

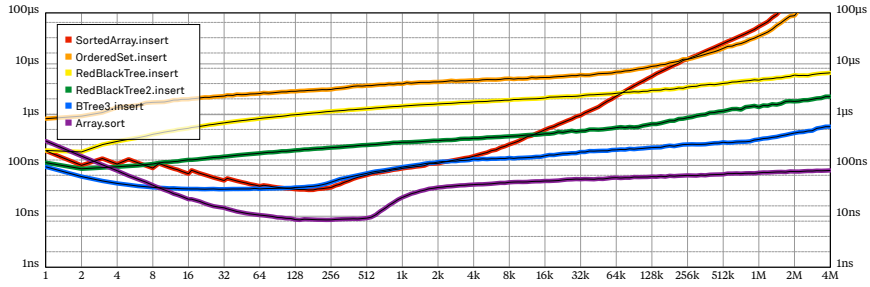


图 8.1: 对比五种 **SortedSet** 实现的插入性能与 **Array.sort** 的每个元素的平摊开销。

我们还有可能实现一个更快的 SortedSet.insert 吗？毫无疑问，BTree3 还有一些额外的空间可以进行小幅优化；我想 5-10% 改善完全不是问题。如果我们投入更多努力，甚至有可能可以得到 20% 的提升。

那是否还存在可以给我们带来 200-400% 的巨大性能跳跃的优化手段，而我们又恰恰错过了这些很巧妙的方法呢？我相信答案是没有。

首先要注意，当我们将一系列元素插入到一个有序集合中时，我们实际上是在对它们进行排序。我们可以简单地调用 Array.sort 来做这件事，这个函数使用了超级快的内省排序算法。在上面的图表中，最后一行描述的就是 Array.sort 在每一个元素上所花费的平摊时间。毫无疑问，Array.sort 为我们能够想到的所有有序集合设定了一个性能上限。

在一般尺寸的集合中，通过调用 BTree3.insert 来进行元素排序相较于 Array.sort **仅仅只慢 3.5 倍**。这个结果简直接近的**让人惊讶**！要知道 BTree3 的性能测试是针对每个元素单独处理的，而且每次插入后所有存在的元素依然要保持有序。对于 BTree3.insert 能够在如此不利的条件下还能表现地这么接近，我真是又惊又喜，如果有一种新的 SortedSet 实现能够将 B 树的性能再提升哪怕 50%，我都会觉得大为震惊。

实现常数时间的插入

虽然可能在保持 BTree3 满足 SortedSet 所有要求的同时，大幅提升性能可能是难以做到的，不过如果我们作点弊的话，我们总归是能让它变得更快的。

比如说，下面代码中的 SillySet 在语法上实现了 SortedSet 协议的要求，而且拥有一个 $O(1)$ 时间复杂度的 insert 方法。它在上面的 insert 性能测试中不费吹灰之力地就能和 Array.sort 一较高下：

```
struct SillySet<Element: Hashable & Comparable>: SortedSet, RandomAccessCollection {
  typealias Indices = CountableRange<Int>

  class Storage {
    var v: [Element]
    var s: Set<Element>
    var extras: Set<Element> = []

    init(_ v: [Element]) {
      self.v = v
      self.s = Set(v)
    }

    func commit() {
      guard !extras.isEmpty else { return }
      s.formUnion(extras)
      v += extras
      v.sort()
      extras = []
    }
  }

  private var storage = Storage([])

  var startIndex: Int { return 0 }

  var endIndex: Int { return storage.s.count + storage.extras.count }
```

```
  // 复杂度：`O(n*log(n))`，此处 `n` 是从上一次 `subscript` 被调用以来插入被调用的次数。
  subscript(i: Int) -> Element {
    storage.commit()
    return storage.v[i]
  }

  // 复杂度：O(1)
  func contains(_ element: Element) -> Bool {
    return storage.s.contains(element) || storage.extras.contains(element)
  }

  // 复杂度：除非存储是共享的，否则为 O(1)
  mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪   Element) {
    if !isKnownUniquelyReferenced(&storage) {
      storage = Storage(storage.v)
    }
    if let i = storage.s.index(of: element) { return (false, storage.s[i]) }
    return storage.extras.insert(element)
  }
}
```

当然了，这里的代码有很多问题：比如，要求 Element 是可哈希的，违反了 Collection 的下标要求，使用 $O(n \log n)$ 这样慢到令人惊讶的时间复杂度等。但是我认为最让人恼火的是，SillySet 的索引下标带有副作用，它将会改变底层存储，这破坏了 Swift 中我们关于值语义含义的假设。(举个例子，在线程中传递 SillySet 是很危险的，即使是只读的并行访问也会导致数据发生竞争。)

这个特定的例子也许看起来实在是很蠢，不过通过将连续插入操作的值收集起来放到一个单独的缓冲区里，以此推迟实际操作的执行时机这个想法本身还是值得赞赏的。将一系列元素用循环的方式一个个插入到有序集合中，这是一种效率很低的做法。我们可以先将这些元素放到单独的缓冲区中排序，然后在线性时间内用一个特殊的批量加载初始化方法将该缓冲区转为一棵 B 树，这么做的话会快很多。

我们不去关心一个有序集合在插入时的中间状态是否满足要求，因为我们绝不会去使用一个只加载到一半的集合，批量加载之所以可行，正是利用了这一点，我们也因此得到了性能的提升。

辨识这类优化机会是十分重要的，它让我们能**暂时**脱离一般的限制，而且这会为我们带来在被限制时所不可能得到的大幅度的性能提升。

再会

我在写这本书的时候获得了很多乐趣，所以我希望您也能喜欢这本书！这一路走来，我学会了不少在 Swift 中实现集合类型的方法，希望您也掌握了一些新的技巧。

在整本书中，我们探索了数种方式，为了解决构建有序集合这一特定问题，我们集中精力对解决方案进行了性能测试，并不断找寻能够改善性能的方式。

不过，这些实现方式都还不完整，我们的代码也并没有真正好到可以在实际产品中使用。为了保持整本书**相对**较短，我们走了一些捷径，而有些捷径并不适合使用在实际项目之中。

就算是我们的 SortedSet 协议，也是被简化至最小的：我们去掉了 SetAlgebra 中大部分的方法。比如，我们从来没有讨论过如何实现一个 remove 操作。可能你会觉得意外，但事实就是，从一个已经平衡的树中移除元素往往要比添加元素困难得多。(你可以自己试试看！)

我们没有花时间去检查那些我们可以通过平衡搜索树构建的其他数据类型。树结构的有序映射 (map)、列表 (list)，以及像是多重集合 (multiset) 和多重映射 (multimap)，都是和有序集合一样重要的数据结构；仔细研究如何改写我们的代码来实现它们会是一件非常有意思的事情。

我们也没有对这些实现方式应当如何被测试做出解释。省略这部分是一个非常艰难的决定，因为我们写了不少取巧的代码，有时候我们甚至使用了不安全的构造方式，最微小的错误都可能会酿成可怕的内存错误，造成让人抓狂的调试工作。

测试实在是太重要了；特别是单元测试，它能为集合类型的正确性提供安全保障，也几乎是进行所有优化工作的先决条件。数据结构自身的特点使它非常适合进行单元测试：它们的操作所接受的输入很容易生成，产生的输出也都是定义良好、易于验证的。使用像是 SwiftCheck 这样强大的工具，可以很容易地为项目提供完整的测试覆盖。

不过，测试写时复制的实现并不是一件简单的事。即使我们不够仔细，不小心在调用 isKnownUniquelyReferenced 之前创建了强引用，我们的代码依然会给出正确的结果，只不过会比我们所期待的慢上许多。我们通常不会在单元测试中检查这样的性能问题，所以我们需要用特殊的手段衡量代码的性能，从而用简单的方法来捕获这类问题。

相反，如果忘记了在改变共享存储前对它进行复制，我们的代码将会影响到那些持有未改变的复制的变量。这样的操作所破坏的值并不一定会明显地表现在操作的输入或者输出上，在原因和结果之间**有一定距离的非预期行为**是很难被追踪的。就算是我们拥有 100% 的测试覆盖率，一般的输入/输出也不一定会去检测这种情况。因此为了捕获这种错误，我们需要专门为这种情况编写单元测试。

我们简单提到过，通过向一棵搜索树的节点中添加属性来存储节点下的元素个数，我们就可以在 $O(\log n)$ 的时间内寻找到树中第 i 个最小或者最大的元素。实际上这个技巧可以被一般化：对包含任意尺度的元素的搜索树，对其进行信息扩充，可以加速所有的关联二分操作。在算法问题中，扩充一直是一个秘密武器：它让我们可以轻易地解决很多看上去复杂的问题。我们这里没有时间对如何实现扩充树以及如何使用它们来解决问题做更详细的解释了。

现在是这本书该完结的时候了，针对我们的问题，我们已经找到了看起来最好的数据结构，此外，我们已经准备好着手将它构建成一个完整的、产品级的解决方案了。从各种角度来说，这都不会是一件简单的工作：我们已经研究过不少操作了，但是仍然需要继续构建和测试更多的代码，并为它们编写文档！

如果你很喜欢本书，并想要亲自动手尝试将集合类型的代码优化为产品可用的质量，可以看看我在 GitHub 上开源的 BTree 项目。在写作时，这个项目最新的版本中甚至还没有包含我们原先在 B 树代码中所做的某些优化，更不用说第七章里的一些进阶内容了。这个项目还有很大的改进空间，我们也随时欢迎您作出贡献。

这本书是如何创建的

本书是由 *bookie* 生成的，这是一个我用来创建关于 Swift 书籍的工具。(显然 *Bookie* 是出书人 (bookmaker) 的非正式名字，所以名字上来说我觉得简直是完美契合。)

Bookie 是用 Swift 编写的一个命令行工具，它接受 Markdown 文本文件作为输入，然后生成组织良好的 Xcode Playground、GitHub 样式的 Markdown、EPUB、HTML、LaTeX 以及 PDF 文件，同时它还包括一份含有全部源代码的 Swift 包。Bookie 可以直接生成 playground，Markdown 和源代码，对于其他格式，它将在把文本转换为 Pandoc 自己的 Markdown 方言后再使用 Pandoc 进行生成。

为了验证示例代码，bookie 将会把所有 Swift 代码例子提取到一个特殊的 Swift 包中 (以 #sourceLocation 进行细心标注)，并使用 Swift Package Manager 进行构建。之后得到的命令行 app 将会被运行，所有被用来求值的代码将依次运行，并打印返回值。输出将会被分割，每个独立的结果都将被插回打印版书籍中相应的代码行之后：

```
func factorial(_ n: Int) -> Int {
  return (1 ... max(1, n)).reduce(1, *)
}
▶ factorial(4)
  24
▶ factorial(10)
  3628800
```

(在 playground 中，这些输出是动态生成的；但在其他格式里，输出结果将被包括进来。)

和 Xcode 中一样，语法颜色是通过 SourceKit 完成的。SourceKit 使用的是官方的 Swift 语法，所以上下文关键字总是能够被正确高亮：

```
var set = Set<Int>() // "set" 也是定义属性 setter 的关键字
set.insert(42)
▶ set.contains(42)
  true
```

本书电子版使用的字体是 Adobe 的思源黑体。示例代码使用的是 Laurenz Brunner 的 *Akkurat*。

Bookie (暂时还？) 不是一个免费/开源软件，如果你有兴趣在自己的项目中使用它，请直接联系我。