

≡ Algorithms in Swift 3

◀ BST I - 初始化和插入

返回视频 ▶

(<https://www.boxueio.com/series/algorithms-in-swift3/ebook/87>)

(</series/algorithms-in-swift3>)

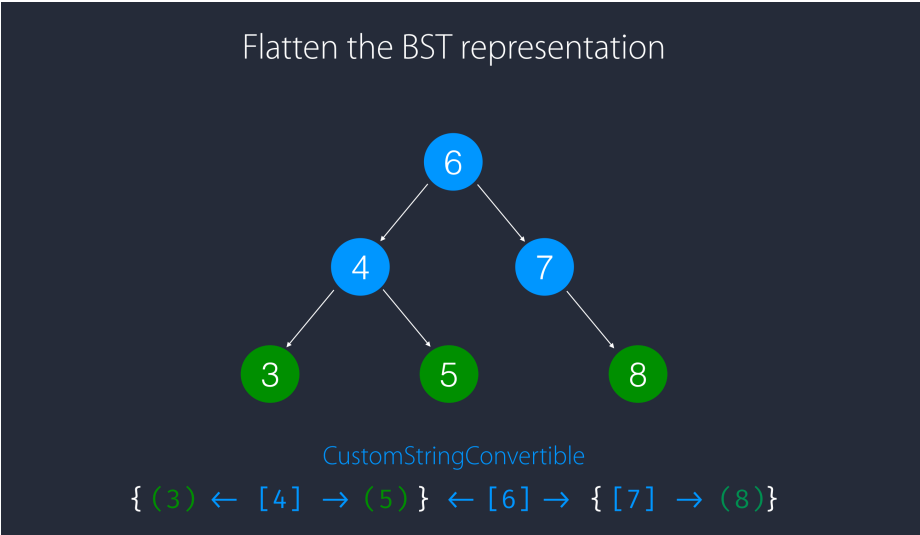
BST II - 打印和遍历

⌂ Back to series (</series/algorithms-in-swift3>)

BST-II

- ⊕ 字号
- 字号
- 🖌️ 默认主题
- 🖌️ 金色主题
- 🖌️ 暗色主题

为了能自定义 print 结果，我们让 BST 实现 CustomStringConvertible protocol 就好了。而打印出来的效果，就像是把 BST 按住root，向下压成一条直线的样子：



- 每个叶子节点用小括号包围；
- 每个包含子树的节点用中括号包围；
- 每个非叶子子树用大括号包围；
- 左右箭头表示 left 和 right ；

为TreeNode添加computed property

为了实现这个功能，我们先在 `TreeNode<T>` 中添加一些判断节点属性的方法：

- `isRoot` : 是否是root；
- `isLeaf` : 是否是叶子；
- `isLeftChild` : 是否是某个节点的左子节点；
- `isRightChild` : 是否是某个节点的右子节点；
- `hasSingleLeftChild` : 是否有单左子节点；
- `hasSingleRightChild` : 是否有单右子节点；
- `hasAnyChild` : 是否有任意一个子节点；
- `hasBothChild` : 是否同时拥有两个子节点；

它们都被实现为 `TreeNode<T>` 的computed property：

```
public var isRoot: Bool {
    return parent == nil
}

public var isLeaf: Bool {
    return left == nil && right == nil
}

public var isLeftChild: Bool {
    return parent?.left === self
}

public var isRightChild: Bool {
    return parent?.right === self
}

public var hasSingleLeftChild: Bool {
    return left != nil && right == nil
}

public var hasSingleRightChild: Bool {
    return left == nil && right != nil
}

public var hasAnyChild: Bool {
    return left != nil || right != nil
}

public var hasBothChild: Bool {
    return left != nil && right != nil
}
```

逻辑都很简单，我们就不一一阐述了。只是在这里强调下 `isLeftChild` 和 `isRightChild` 的实现，我们使用了三个等号来判断两个对象是否为同一对象。

实现打印BST的辅助方法

有了它们之后，接下来，我们在 `BST extension` 中定义一个辅助方法，按照我们定义的规则，把 `BST` 的内容输出到一个字符串：

```

fileprivate func printNode(node: TreeNode<T>?,
    forDebug: Bool = false) -> String {
    guard node != nil else {
        return ""
    }

    var s = ""
    if !node!.isLeaf && !node!.isRoot {
        s += forDebug ? "{ " : "🌱{"
    }

    s += printNode(node: node!.left, forDebug: forDebug)

    let nodeValue = node!.value
    let nodeParentValue = node!.parent?.value

    if node!.isLeaf {
        s += forDebug ? "(🍀:\(nodeValue),P:\(nodeParentValue))"
            : "(\(nodeValue))"
    }
    else if node!.hasSingleLeftChild {
        s += forDebug ? " <- [🌱:\(nodeValue),P:\(nodeParentValue)]"
            : " <- [\(\nodeValue)]"
    }
    else if node!.hasSingleRightChild {
        s += forDebug ? "[🌱:\(nodeValue),P:\(nodeParentValue)] -> "
            : "[\(\nodeValue)] -> "
    }
    else if node!.hasBothChild {
        s += forDebug ? " <- [🌲:\(nodeValue),P:\(nodeParentValue)] -> "
            : " <- [\(\nodeValue)] -> "
    }

    s += printNode(node: node!.right, forDebug: forDebug)

    if !node!.isLeaf && !node!.isRoot {
        s += "}"
    }

    return s
}

```

printNode 接受一个 forDebug 参数用来输出详细程度不同的信息，它的整体实现分成几大部分：

- printNode(node: node!.left) 之前：

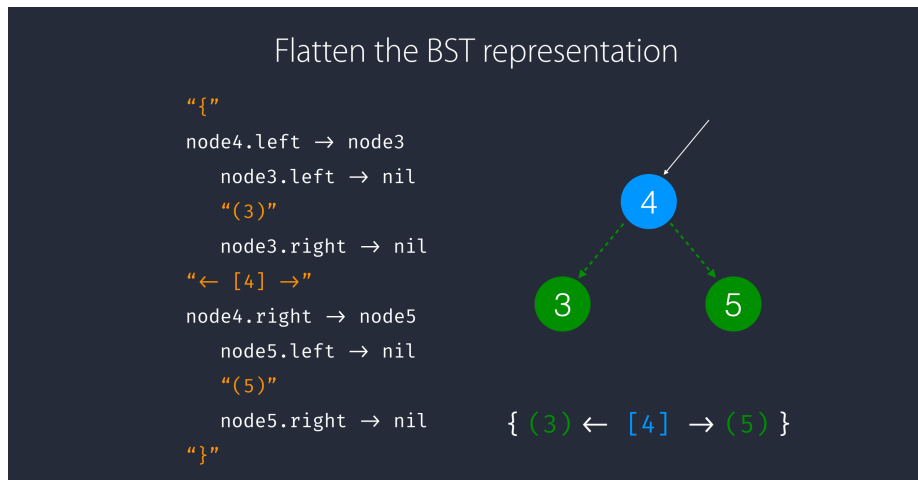
打印每一个 BST 的起始情况。当节点为空时，直接返回，否则，我们就新建一个用于保存输出内容的字符串，如果当前节点不是 root 和 leaf，就输出一个 { 到 s，表示开始输出一个新的子树；

- printNode(node: node!.left) - printNode(node: node!.right)

如何理解这段代码呢？从宏观上看，就是先打印一个节点的左子树部分，再打印节点本身，再打印节点的右子树部分。而从微观看，整个过程一定会“下探”到某个叶子节点才真正开始执行输出。由于叶子节点没有左子树，所以直接输出了叶子节点本身，然后，这个叶子节点的右子树也为空，同样不需要做任何操作。这时，执行就返回到了叶子节点的父节点，输出这个父节点，然后，同理输出这个父节点的右子树部分。周而复始，整个 BST 就被输出成一个字符串了。

- printNode(node: node!.right) 之后：

每当输出完一个节点的右子树时（叶子和根节点除外），我们就可以输出一个 } 表示一个子树输出结束了。整个过程，如下图所示：



在图中，我们可以清楚的看到，先输出一个节点的左子树，再输出节点本身，最后输出节点的右子树的过程。

实现BST的自定义输出

有了 printNode 这个辅助方法之后，执行打印就容易多了，我们分别实现 CustomStringConvertible 和 CustomDebugStringConvertible 这两个 protocol：

```

extension BST: CustomStringConvertible {
    open var description: String {
        return self.printNode(node: self.root, forDebug: false)
    }
}

extension BST: CustomDebugStringConvertible {
    open var debugDescription: String {
        return self.printNode(node: self.root, forDebug: true)
    }
}

```

然后，用一开始我们设计时使用的 BST 来测试下：

```

let tree1 = BST<Int>([6, 4, 7, 3, 5, 8])
print(tree1)
debugPrint(tree1)

```

就可以在控制台看到相应的结果了：

```

{(3) <- [4] -> (5)} <- [6] -> {[7] -> (8)}
{((3, P:Optional(4)) <- [4, P:Optional(6)] -> (5, P:Optional(4))) <- [6, P:nil] -> {([7, P:Optional(6)] -> (8, P:Optional(7)))}
Program ended with exit code: 0

```

进一步抽象打印BST时的思路

抛开输出 BST 时执行的各种细节，我们其实采取了一种既定的顺序，既左子树、树根、右子树。这样的形式，是最贴近我们对树结构的图形化理解的。但不一定对计算机足够友好，因为如果我们想先处理树根，这样的得到的遍历结果就不太方便。

因此，根据对树根的处理顺序，我们有三种遍历BST内容的方式（暂时忽略按层遍历）：

- Preorder: 树根、左子树、右子树；
- Inorder: 左子树、树根、右子树；
- Postorder: 左子树、右子树、树根；

看到这里，你可能会想到了，我们可以把这三种遍历方法，分别定义成 high order function。首先，定义一些辅助方法：

```

fileprivate func preorderTraverse(node: TreeNode<T>?,
    process: (TreeNode<T>) -> Void) {
    guard let node = node else { return }

    process(node)
    preorderTraverse(node: node.left, process: process)
    preorderTraverse(node: node.right, process: process)
}

fileprivate func inorderTraverse(node: TreeNode<T>?,
    process: (TreeNode<T>) -> Void) {
    guard let node = node else { return }

    inorderTraverse(node: node.left, process: process)
    process(node)
    inorderTraverse(node: node.right, process: process)
}

fileprivate func postorderTraverse(node: TreeNode<T>?,
    process: (TreeNode<T>) -> Void) {
    guard let node = node else { return }

    postorderTraverse(node: node!.left, process: process)
    postorderTraverse(node: node!.right, process: process)
    process(node)
}

```

然后，提供对外的遍历方法：

```

public func preorderTraverse(process: (TreeNode<T>) -> Void) {
    self.preorderTraverse(node: self.root, process: process)
}

public func inorderTraverse(process: (TreeNode<T>) -> Void) {
    self.inorderTraverse(node: self.root, process: process)
}

public func postorderTraverse(process: (TreeNode<T>) -> Void) {
    self.postorderTraverse(node: self.root, process: process)
}

```

用下面的代码测试一下：

```

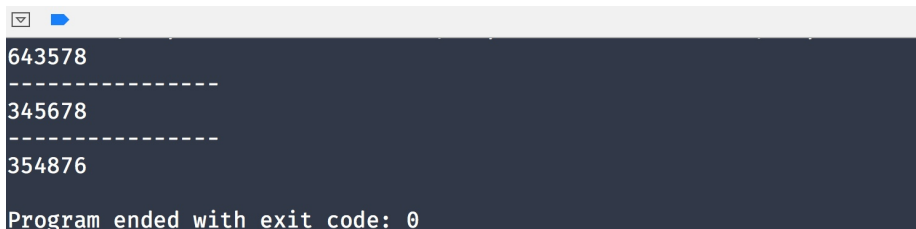
tree1.preorderTraverse {
    print($0.value, terminator: "")
}
print("\n-----")

tree1.inorderTraverse {
    print($0.value, terminator: "")
}
print("\n-----")

tree1.postorderTraverse {
    print($0.value, terminator: "")
}
print("\n")

```

就能在控制台看到对应的遍历结果了：



```

643578
-----
345678
-----
354876
Program ended with exit code: 0

```

Next?

这就是这段视频的全部内容，我们了解了如何创建、插入和遍历 BST，以及一些 Swift 3 中的新特性。在下一段视频中，我们将继续实现 BST 和 `TreeNode` 中的常用属性和方法。

[◀ BST I - 初始化和插入](#)[返回视频 ▶](#)<https://www.boxueio.com/series/algorithms-in-swift3/ebook/87></series/algorithms-in-swift3>

职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表，清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二）(<https://www.boxueio.com/after-the-full-upgrade-to-swift3>)

Mar 4, 2017

人生中第一次创业的"10有"(<https://www.boxueio.com/founder-chat>)

Jan 9, 2016

猎云网采访报道泊学(<http://www.lieyunwang.com/archives/144329>)

Dec 31, 2015

What most schools do not teach(<https://www.boxueio.com/what-most-schools-do-not-teach>)

Dec 21, 2015

一个工作十年PM终创业的故事（一）(<https://www.boxueio.com/founder-story>)

May 8, 2015

泊学相关

[关于泊学](#)

>

[加入泊学](#)

>

[泊学用户隐私及服务条款 \(HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE\)](https://www.boxueio.com/terms-of-service)

[版权声明 \(HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT\)](https://www.boxueio.com/copyright-statement)

联系泊学

Email: 10@boxue.io (<mailto:10@boxue.io>)

QQ: 2085489246

2017 © Boxue, All Rights Reserved. 京ICP备15057653号-1 (<http://www.miibeian.gov.cn/>) 京公网安备 11010802020752号 (<http://www.beian.gov.cn/portal/registerSystemInfo?recordcode=11010802020752>)

友情链接 [SwiftV \(http://www.swiftv.cn/\)](http://www.swiftv.cn/) | [Seay信息安全博客 \(http://www.cnseay.com/\)](http://www.cnseay.com/) | [Swift.gg \(http://swift.gg/\)](http://swift.gg/) | [Laravist \(http://laravist.com/\)](http://laravist.com/) | [SegmentFault \(https://segmentfault.com/\)](https://segmentfault.com/) | [靛青K的博客 \(http://blog.dianqk.org/\)](http://blog.dianqk.org/)