MAP

TreeMap

使用红黑树存储键值对的Map，将Key，Value值封装成一个Node节点保存在红黑树上。查找和保存方式类似于红黑树。

PS： **Key必须具备可比较性**

HashMap

是一段连续的内存（数组）内根据传入的Key的hash值进行存储，每个数组的节点内存放目标值的结构，当产生哈希冲突的时候节点内的值会变为单链表或红黑树进行存储。是一种空间换时间的数据结构

PS： **Key可以不具备可比较性，但必须可以被哈希**

哈希表的实现步骤

1. 生成Key的哈希值，Int型
2. 将哈希值和数组长度进行运算生成索引值。
3. 将Key和Value封装成Node节点放入数组的该索引中
4. 不同的Key可能会出现哈希冲突，按“哈希冲突”方法解决

哈希冲突

哈希冲突是两个不同的Key哈希后的值相等或者经过计算放置在数组的同一位置的情况。Java解决哈希冲突的方法是链地址法。默认由单向链表将冲突的元素串起来（**使用单向链表而不是双向链表是因为添加删除和查找需要遍历每一个元素来确定是否存在相同的Key，不增加Prev指针节省内存空间**），当连接数量过多时会转成红黑树存储。

Hashable和Equatable

在Swift中，Hashable协议继承自Equatable，如果使用自定义对象作为Key同时需要实现 `static func == (lhs: Person, rhs: Person) -> Bool` 方法和 `var hashValue: Int` 或者 `func hash(into hasher: inout Hasher)` 当一个Key相等时，哈希值一定相同，哈希值相同时， 两个Key不一定相同。

‘ == ’ 的几个性质：

1. 自反性： a == a 一定返回true
2. 对称性 a == b 返回true的话。b == a 也是true
3. 传递性： a== b 返回true ， b == c 也是true的话， 那么 a == c 一定是true

实现

- 做hash算法的时候可以根据key.hashValue进行位运算，因此可以扩展Int的 >>> 无符号右移操作符。

```
private func hash(key: AnyHashable?) -> Int {
        let hash = key == nil ? 0 : key.hashValue
        return hash ^ (hash >>> 16)
}

/// 根据key生成对应的索引（在桶数组中的位置）
    /// - Parameter key: <#key description#>
    /// - Returns: <#description#>
    private func getIndex(key: AnyHashable) -> Int {
        return hash(key: key) & (table.count - 1)
    }

infix operator >>> :  BitwiseShiftPrecedence
extension Int {
    static func >>>(lhs:Int, rhs: Int) -> Int {
        if lhs >= 0 {
                return lhs >> rhs
            } else {
                return (Int.max + lhs + 1) >> rhs | (1 <<
(63-rhs))
            }
    }
}
```

- 添加节点的时候参考红黑树的添加，在每次添加的时候可以计算是否需要扩容，增加查找效率。这里用了比较笨的遍历删除在重新添加的方法。也可以遍历每个元素重新位移，因为元素只有不变和2^n符号位从0 -> 1两种可能性。效率更高。

```
private func resize() {
        if Double(size) / Double(table.count) <=
loadFactor { return }

        var oldTable = [HashMapNode<K,V>?]()

        for i in 0..<table.count {
            oldTable.append(table[i])
        }

        table.removeAll()
        for _ in 0..<oldTable.count * 2 {
            table.append(nil)
        }

        let queue = Queue<HashMapNode<K,V>>()
        for i in 0..<oldTable.count {
            if oldTable[i] == nil { continue }
            queue.offer(oldTable[i]!)

            while !queue.isEmpty() {
                let node = queue.poll()
                if node?.left != nil {
                    queue.offer(node!.left!)
                }

                if node?.right != nil {
                    queue.offer(node!.right!)
                }

                moveNode(node!)
            }
        }

    }

    private func moveNode(_ newNode: HashMapNode<K,V>) {
        newNode.left = nil
        newNode.right = nil
        newNode.parent = nil
        newNode.color = .Red

        put(newNode.key, value: newNode.value)
    }
```

- put时候的比较和红黑树不同,首先根据哈希值来判断放在父节点的左边或者右边（因为不强制实现comparable协议，所以Key可能无法使用 '>' '<' 符号比较大

小），哈希值相等的情况下判断key相等覆盖原节点，否则遍历查找是否有相同的Key，search标记保证遍历只执行一次。如果没有相同的Key（这里选择了）随机放到遍历到的最后一个叶子节点的左或者右（也可以根据内存地址大小等其他手段放置到左右子节点）

```
while currentNode != nil {
            parent = currentNode
            let key2 = currentNode!.key
            let h2 = hash(key: key2)
            if h1 > h2 {
                result = .orderedAscending
            } else if (h1 < h2) {
                result = .orderedDescending
            } else if (key == key2) {
                result = .orderedSame
            } else if (searched) {
                // 已经扫描过了，随机放到左侧或右侧
                result = getRandom()
            }else {
                if (currentNode!.left != nil &&
currentNode!.left!.key == key) {
                    result = .orderedSame
                    currentNode = currentNode!.left
                }else if currentNode?.right != nil &&
currentNode!.right!.key == key {
                    result = .orderedSame
                    currentNode = currentNode!.right
                }else {
                    // 已经搜索过了没搜到，随机放到左侧或者右侧
                    searched = true
                    result = getRandom()
                }
            }
            if result == .orderedAscending {
                currentNode = currentNode?.right
            }else if result == .orderedDescending {
                currentNode = currentNode?.left
            }else {
                currentNode?.key = key
                currentNode?.value = value
                return
            }
        }

        let newNode = createNode(key, value: value,
parent: parent)
        if result == .orderedAscending {
```

```
            parent?.right = newNode
        }else {
            parent?.left = newNode
        }
```

- get方法将Key的哈希值算出数组的索引后，拿到数组的该索引的红黑树根节点查找目标Key，查找方式类似于put，先比较两个key的哈希值，然后遍历查找每一个节点。没有找到返回nil。

```
if h1 > h2 {
                result = .orderedAscending
            } else if (h1 < h2) {
                result = .orderedDescending
            } else if (key == key2) {
                result = .orderedSame
            }else {
                if (currentNode?.left != nil &&
currentNode?.left?.key == key) {
                    result = .orderedSame
                    currentNode = currentNode?.left
                }else if currentNode?.right != nil &&
currentNode?.right?.key == key {
                    result = .orderedSame
                    currentNode = currentNode?.right
                }
            }
            if result == .orderedAscending {
                currentNode = currentNode?.right
            }else if result == .orderedDescending {
                currentNode = currentNode?.left
            }else {
                return currentNode!
            }
```

- remove 也是找到该节点key的索引值上的红黑树，和红黑树删除同样的逻辑。这里没有缩容操作，Java中默认已经扩容到n大小的数组不需要缩容，因为随时可能再次扩大到n。