# 二叉树



## 二叉树的性质

1. 二叉树的第i层至多有2^(i - 1)个节点。
2. 深度为k的二叉树至多有2^k - 1个节点。
3. 深度为k，且节点数量为2^k - 1的二叉树为满二叉树。

type TreeNode struct {

Val int

Left \*TreeNode

Right \*TreeNode

}

## 前序遍历

根->左->右

1 2 4 5 7 8 3 6

func preOrderTraverse(root \*TreeNode) []int {

if nil == root { return nil }

ret := []int{}

ret = append(ret, root.Val)

ret = append(ret, preOderTraverse(root.Left)…)

ret = append(ret, preOderTraverse(root.Right)…)

return ret

}

func preOrderTraverse(root \*TreeNode) []int {

if nil == root {

return nil

}

ret := []int{}

t := root

treeList := list.New()

for nil != t || 0 != treeList.Len() {

for nil != t {

ret = append(ret, t.Val)

treeList.PushBack(t)

t = t.Left

}

if 0 != treeList.Len() {

elm := treeList.Back()

t = elm.Value.(\*TreeNode)

t = t.Right

treeList.Remove(elm)

}

}

return ret

}

## 中序遍历

左->根->右

4 2 7 5 8 1 3 6

func midOrderTraverse(root \*TreeNode) []int {

if nil == root { return nil }

ret := []int{}

ret = append(ret, midOrderTraverse(root.Left)…)

ret = append(ret, root.Val)

ret = append(ret, midOrderTraverse(root.Right)…)

return ret

}

func midOrderTraverse(root \*TreeNode) []int {

if nil == root { return nil }

t := root

treeList := list.New()

for nil != t || 0 != treeList.Len() {

for nil != t {

treeList.PushBack(t)

t = t.Left

}

if 0 != treeList.Len() {

elm := treeList.Back()

t = elm.Value.(\*TreeNode)

ret = append(ret, t.Val)

t = t.Right

treeList.Remove(elm)

}

}

return ret

}

## 后序遍历

左->右->根

4 7 8 5 2 6 3 1

func postOrderTraverse(root \*TreeNode) []int {

if nil == root { return nil }

ret := []int{}

ret = append(ret, postOrderTraverse(root.Left)...)

ret = append(ret, postOrderTraverse(root.Right)...)

ret = append(ret, root.Val)

return ret

}

func postOrderTraverse(root \*TreeNode) []int {

if nil == root { return nil }

ret := []int{}

treeList1 := list.New()

treeList2 := list.New()

t := root

for nil != t || 0 != treeList1.Len() {

for nil != t {

treeList1.PushFront(t)

treeList2.PushFront(t)

t = t.Right

}

if 0 != treeList1.Len() {

elm := treeList1.Front()

t = elm.Value.(\*TreeNode)

t = t.Left

treeList1.Remove(elm)

}

}

for 0 != treeList2.Len() {

elm := treeList2.Front()

t = elm.Value.(\*TreeNode)

ret = append(ret, t.Val)

treeList2.Remove(elm)

}

return ret

}

## 层次遍历

func breadthTraverse(root \*TreeNode) [][]int {

if nil == root { return nil }

ret := [][]int{}

t := root

treeList1 := list.New()

treeList2 := list.New()

treeList1.PushBack(t)

for 0 != treeList1.Len() || 0 != treeList2.Len() {

line := []int{}

for 0 != treeList1.Len() {

elm := treeList1.Front()

t = elm.Value.(\*TreeNode)

line = append(line, t.Val)

if nil != t.Left { treeList2.PushBack(t.Left) }

if nil != t.Right { treeList2.PushBack(t.Right) }

treeList1.Remove(elm)

}

if len(line) > 0 { ret = append(ret, line) }

line = []int{}

for 0 != treeList2.Len() {

elm := treeList2.Front()

t = elm.Value.(\*TreeNode)

line = append(line, t.Val)

if nil != t.Left { treeList1.PushBack(t.Left) }

if nil != t.Right { treeList1.PushBack(t.Right) }

treeList2.Remove(elm)

}

if len(line) > 0 { ret = append(ret, line) }

}

return ret

}

## 二叉树最大深度

func treeMaxDepth(root \*TreeNode) int {

if nil == root {

return 0

}

return int(math.Max(float64(treeMaxDepth(root.Left)), float64(treeMaxDepth(root.Right)))) + 1

}

## 二叉树最小深度

func treeMinDepth(root \*TreeNode) int {

if nil == root {

return 0

}

left := treeMinDepth(root.Left)

right := treeMinDepth(root.Right)

if 0 != left && 0 != right {

return minInt(left, right) + 1

}

return maxInt(left, right) + 1

}

## 二叉树节点个数

func treeNodeNum(root \*TreeNode) int {

if nil == root {

return 0

}

return treeNodeNum(root.Left) + treeNodeNum(root.Right) + 1

}

## 二叉树第K层的节点个数

求以root为根的k层节点数目，等价于求root左节点为根的第k-1层和root右节点为根的第k-1层的节点数目之和。

func KthLvNodeNum(root \*TreeNode, k int) int {

if nil == root {

return 0

}

if 1 == k {

return 1

}

return KthLvNodeNum(root.Left, k - 1) + KthLvNodeNum(root.Right, k - 1)

}

## 判断两个二叉树是否相同

func isSameTree(rootA \*TreeNode, rootB \*TreeNode) bool {

if nil == rootA && nil == rootB {

return true

}

if nil == rootA || nil == rootB {

return false

}

if rootA.Val != rootB.Val {

return false

}

return isSameTree(rootA.Left, rootB.Left) && isSameTree(rootA.Right, rootB.Right)

}

## 判断是否树的子结构

func isSub (root1 \*TreeNode, root2 \*TreeNode) bool {

if nil == root2 {

return true

}

if nil == root1 {

return false

}

if root1.Val != root2.Val {

return false

}

return isSub(root1.Left, root2.Left) && isSub(root1.Right, root2.Right)

}

func isSubTree(root1 \*TreeNode, root2 \*TreeNode) bool {

if nil == root1 || nil == root2 {

return false

}

var ret bool

if root1.Val == root2.Val {

ret = isSub(root1, root2)

}

if !ret {

ret = isSub(root1.Left, root2)

}

if !ret {

ret = isSub(root1.Right, root2)

}

return ret

}

## 根据前序遍历和中序遍历构建二叉树

func constructTree(startPre int, endPre int, startIn int, endIn int, preOder []int, inOder []int) \*TreeNode {

root := &TreeNode{

Val : preOder[startPre],

Left : nil,

Right : nil,

}

if startPre == endPre {

if startIn == endIn && preOder[startPre].Val == inOder[startIn].Val {

return root

} else {

return nil

}

}

rootIn := startIn

for rootIn <= endIn && inOder[rootIn] != root.Val {

rootIn++

}

if rootln == endln && inOder[rootIn] != root.Val {

return nil

}

//中序遍历中找到根节点之后，其左边就是左子树，右边就是右子树

lenLeft := rootIn - startIn

if lenLeft > 0 {

root.Left = constructTree(startPre + 1, startPre + lenLeft, startIn, rootln, preOder, inOder)

}

if lenLeft < endPre - startPre {

root.Right = constructTree(startPre + lenLeft + 1, endPre, rootIn + 1, endIn, preOder, inOder)

}

}

func construct(preOder []int, inOder []int) \*TreeNode {

len1, len2 := len(preOder), len(inOder)

if 0 == len1 || 0 == len2 {

return nil

}

return constructTree(0, len1 - 1, 0, len2 - 1, preOder, inOder)

}

## 对称二叉树

func isSymTree(r1 \*TreeNode, r2 \*TreeNode) bool {

if nil == r1 && nil == r2 {

return true

}

if nil == r1 || nil == r2 {

return false

}

if r1.Val != r2.Val {

return false

}

return isSymTree(r1.Left, r2.Right) && isSymTree(r1.Right, r2.Left)

}

func isSymmetric(root \*TreeNode) bool {

if nil == root {

return true

}

return isSymTree(root, root)

}

## 二叉树的镜像

func mirrorRecursively(root \*TreeNode) {

if nil == root {  
 return

}

if nil == root.Left && nil == root.Right {

return

}

tmpNode := root.Left

root.Left = root.Right

root.Right = tmpNode

if nil != root.Left {

mirrorRecursively(root.Left)

}

if nil != root.Right {

mirrorRecursively(root.Right)

}

}

## 二叉树的最近公共先祖

func lowestCommonAncestor(root, p, q \*TreeNode) \*TreeNode {

if nil == root || p.Val == root.Val || q.Val == root.Val {

return root

}

left := lowestCommonAncestor(root.Left, p, q)

right := lowestCommonAncestor(root.Right, p, q)

if nil != left && nil != right {

return root

} else if nil == left {

return right

}

return left

}

## 二叉树的最大路径和

func treePathSum(root \*TreeNode, sum \*int) int {

if nil == root {

return 0

}

left := maxInt(treePathSum(root.Left, sum), 0)

right := maxInt(treePathSum(root.Right, sum), 0)

\*sum = maxInt(left + right + root.Val, \*sum)

return maxInt(left, right) + root.Val

}

func maxPathSum(root \*TreeNode) int {

if nil == root {

return 0

}

sum := math.MinInt32

treePathSum(root, &sum)

return sum

}

# 二叉查找（搜索）树

对于树中任意一个节点X，其左子树中任意节点的值都小于X的值，其右子树中任意节点的值都大于X。

## 判断是否二叉查找树

二叉查找树的中序遍历是一个递增的数组，这个特性也被用于判断一颗树是不是二叉查找树。

func isBST(root \*TreeNode, treeArray \*[]int) {

if nil == root {

return

}

isBST(root.Left, treeArray)

\*treeArray = append(\*treeArray, root.Val)

isBST(root.Right, treeArray)

}

func checkBST(root \*TreeNode) bool {

treeArray := []int{}

isBST(root, &treeArray)

if 0 == len(treeArray) {

return true

}

for i:= 1; i < len(treeArray); i++ {

if treeArray[i] <= treeArray[I - 1] {

return fasle

}

}

return true

}

另一种做法则是大小值的做法，对于树中任一一个节点，其左子树的最大值是该节点的值。其右子树的最小值是该节点的值。

func isBST(root \*TreeNode, min int, max int) bool {

if nil == root {

return true

}

if root.Val <= min || root.Val >= max {

return false

}

return isBST(root.Left, min, root.Val) && isBST(root.Right, root.Val, max)

}

func checkBST(root \*TreeNode) bool {

return isBST(root, math.MinInt64, math.MaxInt64)

}

## 二叉搜索树的最近公共先祖

func lowestCommonAncestor(root, p, q \*TreeNode) \*TreeNode {

if nil == root { return nil }

if p.Val < root.Val && q.Val < root.Val {

return lowestCommonAncestor(root.Left, p, q)

} else if p.Val > root.Val && q.Val > root.Val {

return lowestCommonAncestor(root.Right, p, q)

}

return root

}

# 平衡二叉树（AVL树）

空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

## 判断是否为平衡二叉树

func isAVLTree(root \*TreeNode) bool {

if nil == root {

return true

}

if int(math.Abs(float64(treeDepth(root.Left) - treeDepth(root.Right)))) > 1 {

return false

}

return isAVLTree(root.Left) && isAVLTree(root.Right)

}

# 满二叉树

一个二叉树的节点要么是叶子节点，要么有两个叶子节点，被称作满二叉树。

层数为K的满二叉树，其节点个数为2^k - 1。

# 完全二叉树

层数为K的二叉树，除了第K层以外，其他各层节点个数都达到最大，第K层节点都连续集中在数的左边，则成为完全二叉树。

## 判断是否完全二叉树

func isCompleteTree(root \*TreeNode) bool {

if nil == root { return true }

t := root

s := list.New()

cnt := 1

s.PushBack(t)

nilLeaf := false

for 0 != s.Len() {

nextLv := 0

for cnt > 0 {

elm := s.Front()

t = elm.Value.(\*TreeNode)

cnt--

if nilLeaf && (nil != t.Left || nil != t.Right) { return false }

if nil != t.Left {

s.PushBack(t.Left)

nextLv++

}

if nil != t.Right {

s.PushBack(t.Right)

nextLv++

}

s.Remove(elm)

if nil == t.Left && nil != t.Right { return false }

if nil == t.Left || nil == t.Right { nilLeaf = true }

}

cnt = nextLv

}

return true}

# 红黑树

为什么有了二叉查找树和平衡二叉树还需要红黑树？

二叉查找树没有平衡性保证，在极端情况下二叉查找树的节点都在根节点的左边或者右边，退化成一个链表。查找效率低下。

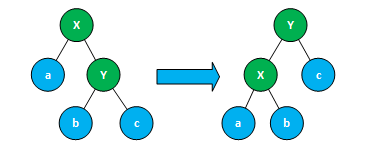
平衡二叉树虽然解决了二叉查找树平衡性的问题，但是平衡二叉树在插入节点的时候为了满足平衡二叉树的特性，需要通过左旋或者右旋来调整整个树，导致插入性能很低。

红黑树各类操作最差的时间复杂度为O(logn)。

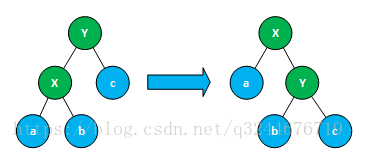
红黑树是一棵平衡二叉查找树，同时还多出如下几点要求：

1. 每个节点要么是红的要么是黑的。
2. 根节点是黑的。
3. 每个叶子节点都是黑的。
4. 如果一个节点是红的，其两个子节点都是黑的。
5. 对于任意节点而言，其到任意尾端为NIL的叶子节点的路径中包含相同数目的黑节点。

左旋



右旋



自平衡策略

左旋、右旋、变色

1. 对于当前节点如果右子节点是红色，左子节点是黑色，就要左旋。
2. 对于当前节点如果左子及左孙子节点都是红色的，就要右旋。
3. 对于当前节点如果左右子节点都是红色，就要变色。

红黑树默认插入的节点是红色的。

1. 插入的节点是根节点，违反性质2，因此只需要将其变为黑色。
2. 插入的节点的父节点是黑色，什么都不需要做，没有违反任何性质。
3. 插入节点的父节点是红色，且祖父节点的另一个子节点是红色。将当前节点的父节点和祖父节点的另一个子节点改为黑色，祖父节点改为红色。将当前节点指向祖父节点，重新开始调整。
4. 当前节点的父节点是红色，叔叔节点是黑色，当前节点是其父节点的右子。将当前节点的父节点作为新的当前节点，并以其进行左旋。
5. 当前节点的父节点是红色，叔叔节点是黑色，当前节点是其父节点的左子节点。将父节点变为黑色，祖父节点变为红色，以祖父节点进行右旋。

# B树/B+树

B树是一种平衡多叉树，B树的定义有两种方式，阶或者度。

B树的阶是指所有节点孩子个数的最大值，用m表示，m>=2。

根节点至少有2个孩子。

每个中间节点都有k-1个元素和k个孩子，其中m/2 <= k <= m。

每个叶子节点都包含k-1个元素，其中m/2 <= k <= m。

所有的叶子节点都位于同一层。

B树的最小度数是指内节点中节点最小孩子数目，用t表示，t>=2。

每个非根节点至少有t-1个关键字，最多可以包含2t-1个关键字。

阶和度都是在创建一个B树的时候提前设定好的值，根据阶或者度可以判断B树中的某个节点是否已满。

B+树

在B树的特点之上，B+新增的几个特点：

每一个节点的关键字都出现在子节点中，是子节点中最大（或最小）的关键字。根节点最大的关键字，就是整个B+数最大的关键字。后续不论插入删除多少元素，都要保持最大的元素在根节点中。

叶子节点中包含了所有的关键字信息，同时所有的叶子节点又构成了一个有序的链表。

B数每个节点上都有挂载对应的卫星数据，而B+数，卫星数据都挂载在叶子节点上，非叶子节点仅起到索引的作用，没有挂载具体的数据。

B+树比B树的优势在于，由于B+树非叶子节点没有挂载数据，这样相同的磁盘页空间内，可以容纳更多的非叶子节点，在查询的时候，B+树拥有更好的IO性能。

同时B+数比B数更适合范围查询，因为B+数的叶子节点是通过链表相连且有序的，范围查询的时候只要找到范围起点在链表中的位置，然后遍历链表很快就能找到范围终点在链表中的位置。而B树则只能使用中序遍历的方法，效率低。