# 算法与数据结构

## 什么是算法

1. 算法是用于解决特定问题的一系列的执行步骤

public static int plus(int a, int b) {//计算a+b的和

return a+ b;

}

public static int sum2(int n) {//计算1+2+3+....+n的和

return (1 + n) \* n/2 ;

}

public static int sum1(int n) {//计算1+2+3+....+n的和

int result = 0;

for (int i = 0; i < n; i++) {

result += 1;

}

return result;

}

1. 使用不同算法，解决同一个问题，效率可能相差非常大

比如：求第n个斐波那契数（fibonacci number）

# 如何评判一个算法的好坏？

# 如果单从执行效率上进行评估，可能会想到这么一种方案

比较不同算法对同一组输入的执行处理时间

这种方案也叫做：事后统计法

上述方案有比较明显的缺点

执行时间严重依赖硬件以及运行时各种不确定的环境因素

必须编写相应的测算代码

测试数据的选择比较难保证公正性

一般从以下维度来评估算法的优劣

正确性、可读性、健壮性（对不合理输入的反应能力和处理能力）

时间复杂度（time complexity）：估算程序指令的执行次数（执行时间）

空间复杂度（space complexity）：估算所需占用的存储空间

大O表示法（Big O）

一般用大O表示法来描述复杂度， 它表示的是数据规模 n 对应的复杂度

忽略常数、系数、低阶

9 >> O(1)

2n + 3 >> O(n)

n2 + 2n + 6 >> O(n2)

4n3 + 3n2 + 22n + 100 >> O(n3)

写法上， n3 等价于 n^3

注意：大O表示法仅仅是一种粗略的分析模型，是一种估算，能帮助我们短时间内了解一个算法的执行效率

对数阶的细节

对数阶一般省略底数

log2n = log29 ∗ log9n

所以 log2n 、 log9n 统称为 logn



O(1) < O(logn) < O(n) < O(nlogn) < O(n2) < O(n3) < O(2n) < O(n!) < O(nn)

可以借助函数生成工具对比复杂度的大小

<https://zh.numberempire.com/graphingcalculator.php>

算法的优化方向

用尽量少的存储空间

用尽量少的执行步骤（执行时间）

根据情况，可以

空间换时间

时间换空间

# 动态数组

什么是数据结构

数据结构是计算机存储、组织数据的方式

1. 线性结构：线性表（数组，链表，栈，队列，哈希表）
2. 树形结构：二叉树，AVL树，红黑树，B树，堆、Trie，哈夫曼树，并查集
3. 图形结构：邻接矩阵，邻接表

在实际应用中，根据使用场景来选择最合适的数据结构

数组是一种顺序存储的线性表，所有元素的内存地址是连续的

在很多编程语言中，数组都有个致命的缺点 无法动态修改容量

实际开发中，我们更希望数组的容量是可以 动态改变的

# 链表(Linked List)

动态数组有个明显的缺点

可能会造成内存空间的大量浪费

能否用到多少就申请多少内存?

链表可以办到这一点

链表是一种链式存储的线性表，所有元素的内存地址不一定是连续的

反转链表

public ListNode reverseList(ListNode head) {

ListNode newHead = null;

while (head != null) {

ListNode tmp = head.next;

head.next = newHead;

newHead = head;

head = tmp;

}

return head;

}

**一个链表是否有环可以通过快慢指针来判断。**

**虚拟头结点**

有时候为了让代码更加精简，统一所有节点的处理逻辑，可以在最前面增加一个虚拟的头结点(不存储数据)

public LinkedList() {

first = new Node<>(null,null)

}

有时候为了让代码更加精简，统一所有节点的处理逻辑，可以在最前面增加一个虚拟的头结点（不存储数据）

# 双向链表

此前所学的链表，也叫做单向链表

使用双向链表可以提升链表的综合性能

有一个prev 有一个next

**双向链表 vs 单向链表**

粗略对比一下删除的操作数量

单向链表:1 + 2 + 3 + ... + n =(1+n) ∗n/2 = n/2+ n平方/2，除以n平均一下是 1/2 + n/2

双向链表:(1 + 2 + 3 + ... +n/2) \* 2 = n/2 +n平方/4，除以n平均一下是1/2 + n/4

操作数量缩减了近一半

**双向链表 vs 动态数组**

动态数组:开辟、销毁内存空间的次数相对较少，但可能造成内存空间浪费(可以通过缩容解双向链表:开辟、销毁内存空间的次数相对较多，但不会造成内存空间的浪费

如果频繁在尾部进行添加、删除操作，动态数组、双向链表均可选择

如果频繁在头部进行添加、删除操作，建议选择使用双向链表

如果有频繁的(在任意位置)添加、删除操作，建议选择使用双向链表

如果有频繁的查询操作(随机访问操作)，建议选择使用动态数组

有了双向链表，单向链表是否就没有任何用处了?

并非如此，在哈希表的设计中就用到了单链表

至于原因，在哈希表章节中会讲到

# 栈(stack)

栈是一种特殊的线性表，只能在一端进行操作

往栈中添加元素的操作，一般叫做 push，入栈

从栈中移除元素的操作，一般叫做 pop，出栈(只能移除栈顶元素，也叫做:弹出栈顶元素) 后进先出的原则，Last In First Out，LIFO

注意:这里说的“栈”与内存中的“栈空间”是两个不同的概念

# 队列(Queue)

队列是一种特殊的线性表，只能在头尾两端进行操作

队尾(rear):只能从队尾添加元素，一般叫做 enQueue，入队

队头(front):只能从队头移除元素，一般叫做 deQueue，出队

先进先出的原则，First In First Out，FIFO

**运算符优化**

尽量避免使用乘\*、除/、模%、浮点数运算，效率低下

已知n>=0,m>0 n%m等价于n-(m>n?0:m)的前提条件：n < 2m

# 二叉树

**树（tree）的基本概念**

节点、根节点、父节点、子节点、兄弟节点

一棵树可以没有任何节点，称为空树

一棵树可以只有1个节点，也就是根节点

子树、左子树、右子树

**节点的度（degree）：**子树的个数

**树的度：**所以节点度中的最大值

**叶子节点（leaf）：**度为0的节点

**非叶子节点：**度不为0的节点

**层数(level):**根节点在第 1 层，根节点的子节点在第 2 层，以此类推(有些教程也从第 0 层开始计算)

**节点的深度(depth)**:从根节点到当前节点的唯一路径上的节点总数

**节点的高度(height):**从当前节点到最远叶子节点的路径上的节点总数

**树的深度:**所有节点深度中的最大值

**树的高度:**所有节点高度中的最大值

树的深度 等于 树的高度

**二叉树的特点**

每个节点的度最大为 2(最多拥有 2 棵子树)

左子树和右子树是有顺序的

即使某节点只有一棵子树，也要区分左右子树

二叉树是有序树 还是 无序树? 有序树

**二叉树的性质**

非空二叉树的第i层，最多有2i−1 个节点(i ≥ 1)

在高度为h的二叉树上最多有2h − 1个结点(h ≥ 1)

对于任何一棵非空二叉树，如果叶子节点个数为 n0，度为 2 的节点个数为 n2，则有: n0 = n2 + 1

假设度为 1 的节点个数为 n1，那么二叉树的节点总数 n = n0 + n1 + n2

二叉树的边数 T = n1 + 2 \* n2 = n – 1 = n0 + n1 + n2 – 1

因此 n0 = n2 + 1

真二叉树:所有节点的度都要么为 0，要么为 2

满二叉树:最后一层节点的度都为 0，其他节点的度都为 2

假设满二叉树的高度为 h( h ≥ 1 )，那么 第 i 层的节点数量: 2i − 1

叶子节点数量: 2h − 1

总节点数量 n

n= 2h −1=20 +21 +22 +⋯+2h-1

h = log2(n+1)

在同样高度的二叉树中，满二叉树的叶子节点数量最多、总节点数量最多

满二叉树一定是真二叉树，真二叉树不一定是满二叉树

**完全二叉树:**对节点从上至下、左至右开始编号，其所有编号都能与相同高度的满二叉树中的编号对应

叶子节点只会出现最后 2 层，最后 1 层的叶子结点都靠左对齐

完全二叉树从根结点至倒数第 2 层是一棵满二叉树

满二叉树一定是完全二叉树，完全二叉树不一定是满二叉树

**完全二叉树的性质**

度为 1 的节点只有左子树

度为 1 的节点要么是 1 个，要么是 0 个

同样节点数量的二叉树，完全二叉树的高度最小

假设完全二叉树的高度为 h( h ≥ 1 )，那么

至少有2h−1 个节点(20+21+22+⋯+2h−2+1)

最多有2h − 1个节点(20+21+22+⋯+2h−1，满二叉树)

总节点数量为 n

2h−1 ≤n<2h

h −1≤log2n<h

h= floor(log2n) + 1

floor 是向下取整，另外，ceiling 是向上取整

一棵有 n 个节点的完全二叉树(n > 0)，从上到下、从左到右对节点从 1 开始进行编号，对任意第 i 个节点 如果 i = 1 ，它是根节点

如果 i > 1 ，它的父节点编号为 floor( i / 2 )

如果 2i ≤ n ，它的左子节点编号为 2i

如果 2i > n ，它无左子节点

如果 2i + 1 ≤ n ，它的右子节点编号为 2i + 1

如果 2i + 1 > n ，它无右子节点

一棵有 n 个节点的完全二叉树(n > 0)，从上到下、从左到右对节点从 0 开始进行编号，对任意第 i 个节点 如果 i = 0 ，它是根节点

如果 i > 0 ，它的父节点编号为 floor( (i – 1) / 2 ) 如果 2i + 1 ≤ n – 1 ，它的左子节点编号为 2i + 1

如果 2i + 1 > n – 1 ，它无左子节点

如果 2i + 2 ≤ n – 1 ，它的右子节点编号为 2i + 2 如果 2i + 2 > n – 1 ，它无右子节点

**面试题**

**如果一棵完全二叉树有 768 个节点，求叶子节点的个数**

假设叶子节点个数为 n0，度为 1 的节点个数为 n1，度为 2 的节点个数为 n2 总结点个数 n = n0 + n1 + n2，而且 n0 = n2 + 1

n = 2n0 + n1 – 1

完全二叉树的 n1 要么为 0，要么为 1

n1为1时，n = 2n0，n 必然是偶数

叶子节点个数 n0 = n / 2，非叶子节点个数 n1 + n2 = n / 2 ✓n1为0时，n = 2n0 – 1，n 必然是奇数

叶子节点个数 n0 = (n + 1) / 2，非叶子节点个数 n1 + n2 = (n – 1) / 2

叶子节点个数 n0 = floor( (n + 1) / 2 ) = ceiling( n / 2 )

非叶子节点个数 n1 + n2 = floor( n / 2 ) = ceiling( (n – 1) / 2 )

因此叶子节点个数为 384

**二叉搜索树(Binary Search Tree)**

**二叉搜索树是二叉树的一种，是应用非常广泛的一种二叉树，英文简称为 BST**

又被称为:二叉查找树、二叉排序树

任意一个节点的值都大于其左子树所有节点的值

任意一个节点的值都小于其右子树所有节点的值

它的左右子树也是一棵二叉搜索树

**二叉搜索树可以大大提高搜索数据的效率**

**二叉搜索树存储的元素必须具备可比较性**

比如 int、double 等

如果是自定义类型，需要指定比较方式

不允许为 null

**平衡二叉搜索树**

如果从小到大添加节点 二叉搜索树会退化成链表 当n比较大时，两者差异比较大 比如n= 1000000时，二叉搜索树的最低高度是20

删除节点也可能会导致二叉搜索树退化成链表

**平衡**

当节点数量固定时，左右子树的高度越接近，这棵二叉树就越平衡（高度越低）

**理想平衡**

最理想的平衡，就像完全二叉树，满二叉树那样，高度是最小的

**平衡二叉搜索树**

英文简称为:BBST

**经典常见的平衡二叉搜索树有**

AVL树：

Windows NT 内核中广泛使用

红黑树：

C++ STL(比如 map、set )

Java 的 TreeMap、TreeSet、HashMap、HashSet ✓Linux 的进程调度

Ngix 的 timer 管理

**一般也称它们为:自平衡的二叉搜索树(Self-balancing Binary Search Tree)**

**AVL树**

AVL树是最早发明的自平衡二叉搜索树之一

AVL 取名于两位发明者的名字

G. M. Adelson-Velsky 和 E. M. Landis(来自苏联的科学家)

**平衡因子(Balance Factor):**某结点的左右子树的高度差

AVL树的特点

每个节点的平衡因子只可能是 1、0、-1(绝对值 ≤ 1，如果超过 1，称之为“失衡”)

每个节点的左右子树高度差不超过 1

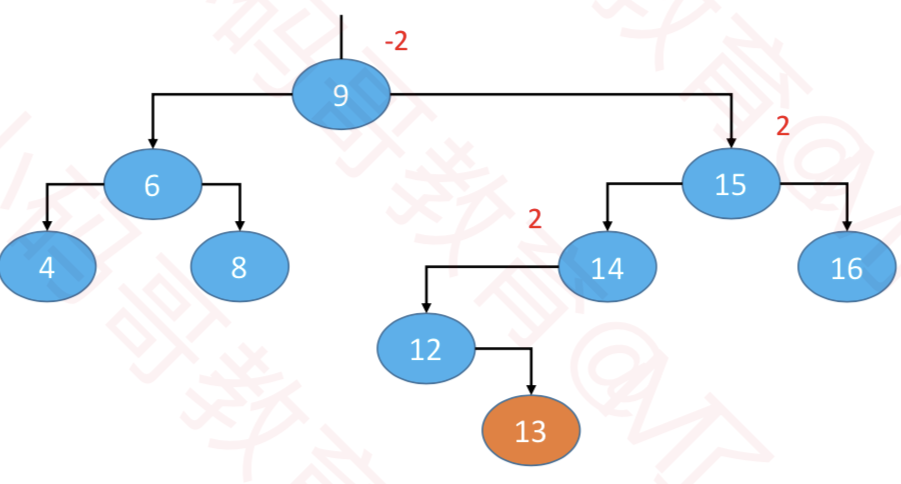
搜索、添加、删除的时间复杂度是 O(logn)

**添加导致的失衡**

示例:往下面这棵子树中添加 13

最坏情况:可能会导致所有祖先节点都失衡

父节点、非祖先节点，都不可能失衡



**LL – 右旋转(单旋)**

g.left = p.right

p.right = g

让p成为这棵子树的根节点

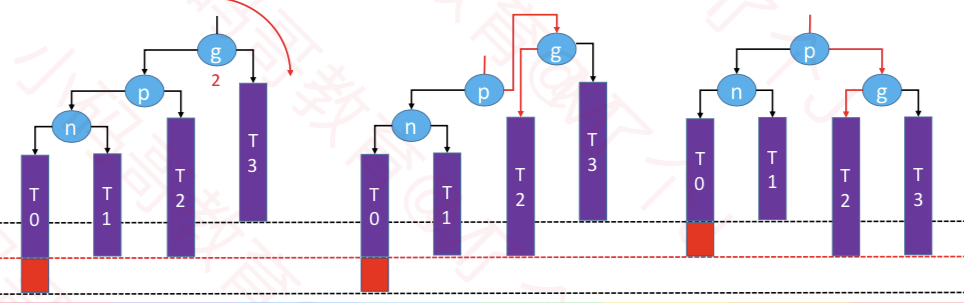
仍然是一棵二叉搜索树:T0 < n < T1 < p < T2 < g < T3

整棵树都达到平衡

**还需要注意维护的内容**

T2、p、g 的 parent 属性

先后更新 g、p 的高度



**RR – 左旋转(单旋)**

g.right = p.left

p.left = g

让p成为这棵子树的根节点

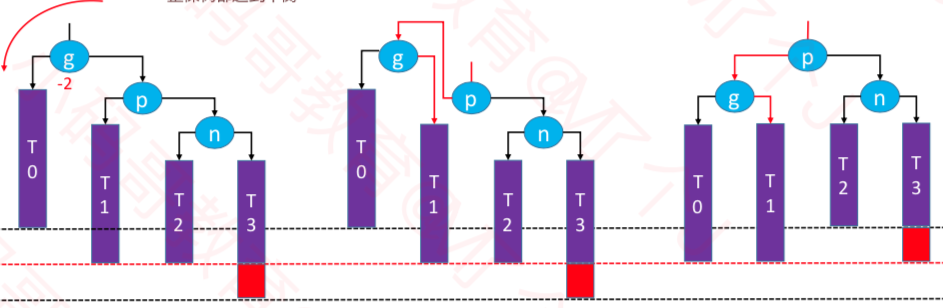
仍然是一棵二叉搜索树:T0 < g < T1 < p < T2 < n < T3

整棵树都达到平衡

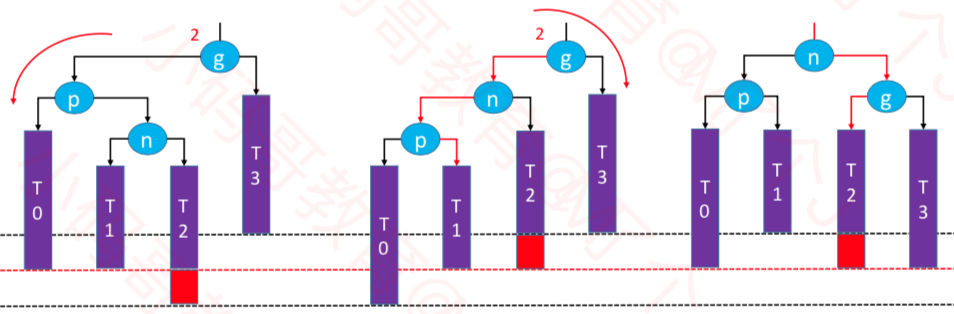
**还需要注意维护的内容**

T1、p、g 的 parent 属性

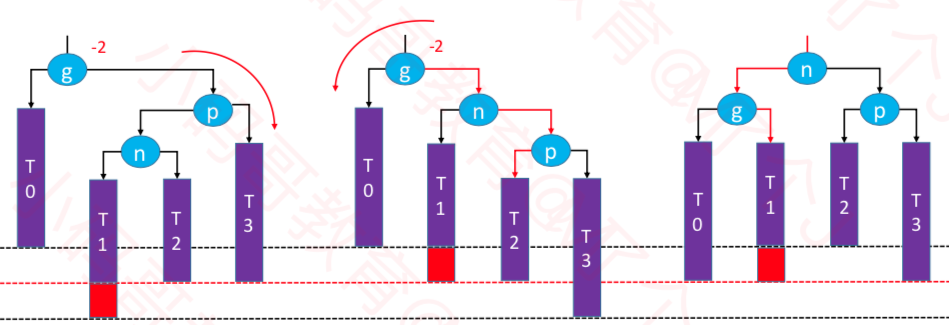
先后更新 g、p 的高度



**LR – RR左旋转，LL右旋转(双旋)**

****

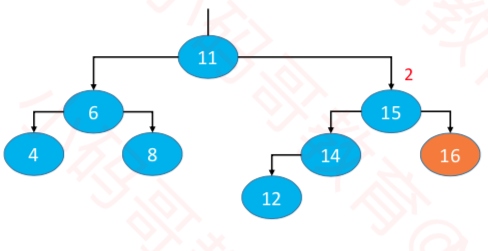
**RL – LL右旋转，RR左旋转(双旋)**

****

**删除导致的失衡**

示例:删除子树中的 16

可能会导致父节点或祖先节点失衡(只有1个节点会失衡)，其他节点，都不可能失衡



**总结**

**添加**

可能会导致所有祖先节点都失衡

只要让高度最低的失衡节点恢复平衡，整棵树就恢复平衡【仅需 O(1) 次调整】

**删除**

可能会导致父节点或祖先节点失衡(只有1个节点会失衡)

恢复平衡后，可能会导致更高层的祖先节点失衡【最多需要 O(logn) 次调整】

**平均时间复杂度**

搜索:O(logn)

添加:O(logn)，仅需 O(1) 次的旋转操作

删除:O(logn)，最多需要 O(logn) 次的旋转操作

**红黑树(Red Black Tree)**

红黑树也是一种自平衡的二叉搜索树

以前也叫做平衡二叉B树(Symmetric Binary B-tree)

红黑树必须满足以下 5 条性质

1. 节点是RED或者BLACK

2. 根节点是BLACK

3. 叶子节点(外部节点，空节点)都是BLACK

4. RED 节点的子节点都是 BLACK

A,RED 节点的 parent 都是 BLACK

B, 从根节点到叶子节点的所有路径上不能有 2 个连续的 RED 节点

1. 从任一节点到叶子节点的所有路径都包含相同数目的BLACK节点

**红黑树的等价变换**

红黑树 和 4阶B树(2-3-4树)具有等价性

BLACK 节点与它的 RED 子节点融合在一起，形成1个B树节点

红黑树的 BLACK 节点个数 与 4阶B树的节点总个数 相等

网上有些教程:用 2-3树 与 红黑树 进行类比，这是极其不严谨的，2-3树 并不能完美匹配 红黑树 的所有情况

**AVL树 vs 红黑树**

**◼AVL树**

**平衡标准比较严格:每个左右子树的高度差不超过1**

**最大高度是 1.44 ∗ log2 n + 2 − 1.328(100W个节点，AVL树最大树高28)**

**搜索、添加、删除都是 O(logn) 复杂度，其中添加仅需 O(1) 次旋转调整、删除最多需要 O(logn) 次旋转调整**

**◼ 红黑树**

平衡标准比较宽松:没有一条路径会大于其他路径的2倍

最大高度是 2 ∗ log2(n + 1)( 100W个节点，红黑树最大树高40)

搜索、添加、删除都是 O(logn) 复杂度，其中添加、删除都仅需 O(1) 次旋转调整

**◼ 搜索的次数远远大于插入和删除，选择AVL树;搜索、插入、删除次数几乎差不多，选择红黑树**

**◼ 相对于AVL树来说，红黑树牺牲了部分平衡性以换取插入/删除操作时少量的旋转操作，整体来说性能要优于AVL树**

**◼ 红黑树的平均统计性能优于AVL树，实际应用中更多选择使用红黑树**

**二叉堆**

堆(Heap)也是一种树状的数据结构(不要跟内存模型中的“堆空间”混淆)，常见的堆实现有 ：

二叉堆(Binary Heap，完全二叉堆)

多叉堆(D-heap、D-ary Heap)

索引堆(Index Heap)

二项堆(Binomial Heap)

斐波那契堆(Fibonacci Heap)

左倾堆(Leftist Heap，左式堆)

斜堆(Skew Heap)

堆的一个重要性质:任意节点的值总是 ≥( ≤ )子节点的值

如果任意节点的值总是 ≥ 子节点的值，称为:最大堆、大根堆、大顶堆

如果任意节点的值总是 ≤ 子节点的值，称为:最小堆、小根堆、小顶堆

由此可见，堆中的元素必须具备可比较性(跟二叉搜索树一样)

二叉堆的逻辑结构就是一棵完全二叉树，所以也叫完全二叉堆

鉴于完全二叉树的一些特性，二叉堆的底层(物理结构)一般用数组实现即可

索引 i 的规律( n 是元素数量)

如果 i = 0 ，它是根节点

如果 i > 0 ，它的父节点的索引为 floor( (i – 1) / 2 )

如果 2i + 1 ≤ n – 1，它的左子节点的索引为 2i + 1

如果 2i + 1 > n – 1 ，它无左子节点

如果 2i + 2 ≤ n – 1 ，它的右子节点的索引为 2i + 2

如果 2i + 2 > n – 1 ，它无右子节点