深入理解数据结构与算法

By 公众号@极客重生

为什么要学习数据结构与算法

- 面试第一道坎（如果想去大厂），掌握常见面试题
- 知道哪些数据结构和算法在实际工作中最常用，最重要
- 目前看到五花八门的新技术，在宏观本质只是“术”的不断丰富，帮助开发人员能够快速高效写应用，一味追求这种新的“术”，不如反过来逐“道”，而数据结构和算法即是“道”。
- 能够快速理解一些开源软件的优秀设计：
  - 数据库存储引擎：MergeTree（clickhouse） vs Delta Tree （TIDB） vs LSM Tree （levelDB，elasticsearch，hbase，OceanBase等新兴数据库） vs Btree （mysql，PostgreSQL，mongodb等传统关系型数据库）
  - 操作系统常用的数据结构：定时器优化(时间轮),进程调度核心CFS算法，路由查找（hash,radix tree, LC-tries等），任务管理（经典的抽象链表），多路复用技术的Epoll的核心结构也是红黑树+双向链表等。
  - 编程语言标准容器： C++STL容器，如map和set都是红黑树实现的， Java的TreeMap，HashMap的底层实现，Go的map底层实现，Rust的TreeMap(采用B树实现)等。
  - redis的各种数据结构设计, nginx用红黑树管理timer等

如何学习数据结构与算法

- **入门学习：**先学习常见数据结构和算法基本知识（书本上）
  - bugfree实现一遍常见的数据结构和算法并写测试case去验证各种情况

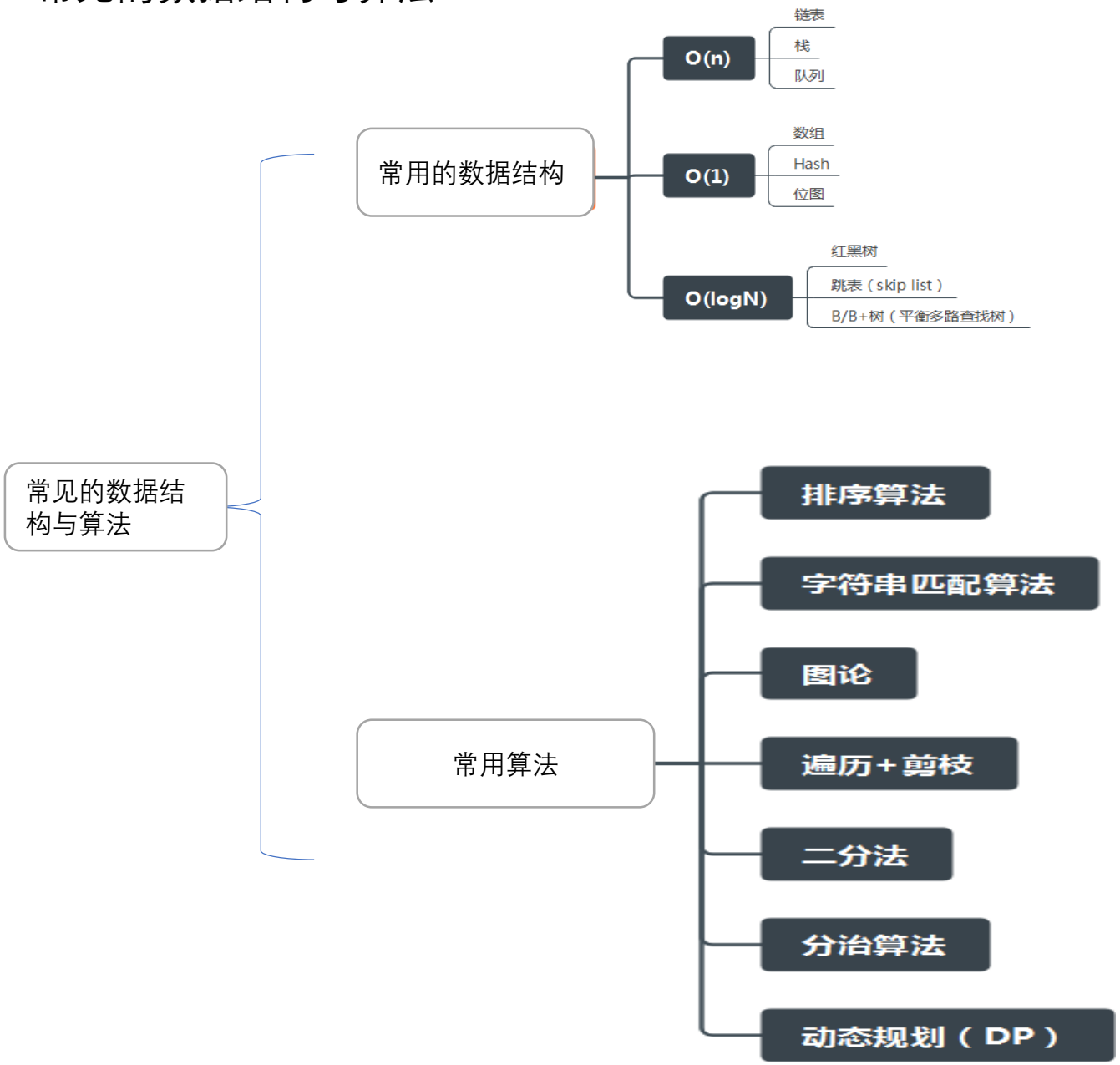
- **刷题应对面试**，保持手感，适当练习白板编程（手写代码）
  - 剑指offer：https://www.nowcoder.com/ta/coding-interviews
  - leetcode：https://leetcode-cn.com/problemset/algorithms/
  - 精选200道leetcode算法题，还有参考答案，分析讲解，先不看答案，认真刷完，并总结回顾，一定可以搞定大厂算法面试:

- **深入学习**：研究开源代码数据结构和算法，深入理解数据结构和算法的本质

- **实战训练**：通过重新设计数据结构或者算法优化程序的性能

By 公众号@极客重生

常见的数据结构与算法



程序 = 算法 +数据结构

By 公众号@极客重生

理解数据结构的本质

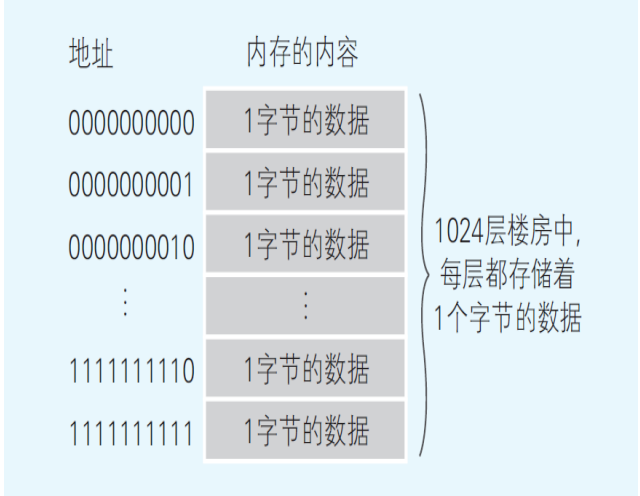
数据结构

- 一个共识：所有计算机程序的最终目的是对数据进行“CRUD”，而CRUD需要通过“查找”“删除”和“插入”3个操作来完成。

程序的本质是对数据（内存或者磁盘）的CRUD，而数据结构是承载数据的容器。

- 一个本质：世界纷繁复杂，如何将世界中的各种数据存储在内存中，并且方便“查找”“删除”和“插入”，就是数据结构的本质。

字节是信息的基本单位，也是最小的“数据结构”



计算机存储的最小单位是一个字节（8个bit），而计算机内存就是一个连续的字节数组。

理解数据结构的本质-数据关系

数据结构分类

- 集合：仅描述元素是否在同一集合中，无其他关系
- 线性结构：元素关系一对一，线性关系，先后关系（包括数组，字符串，线性表、堆栈、队列）
- 树结构：元素关系一对多，层次关系（二叉树，堆，多叉树等）
- 图结构：元素关系多对多，网状结构，可表达元素之间任意关系（线性表、树都是一种特殊的图）

数据结构

数据结构的全景图

抽象

数据库存储引擎（B+/B*,LSM等）

标准库容器（优先队列, set，zmap等）

搜索引擎(字典树 (Trie),并查集,树状数组,线段树等)

性能优化(缓存，无锁队列，时间轮，位图（Bitmap),布隆过滤器等)

高级数据结构

hash

队列

栈

树(BST,堆, AVL , 红黑树 , BTree等)

跳表

图

常见数据结构

数组

字符串

链表

索引

指针

基本数据结构

8位

16位

32位

64位

CPU

基础数据

比特

字节

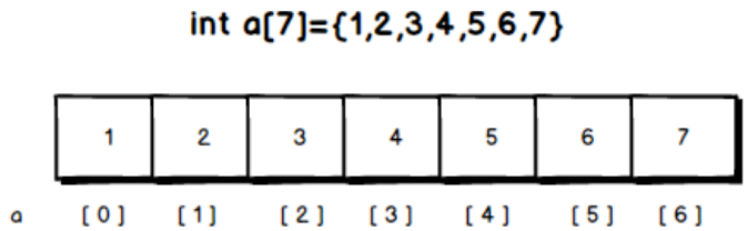
内存

最小数据（软件）

数据结构的基本存储方式就是链式和顺序两种，基本操作就是增删查改，遍历方式无非迭代和递归

理解数据结构-数组

数组



核心点：

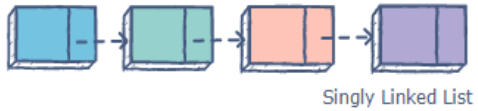
- 内存空间大小固定，如果支持动态扩展，需要内存迁移，有一定的性能代价，比如C++ STL的vector结构；
- 内存连续，对CPU cache友好，如果内存空间足够，能用数组就最好用数组结构；
- 数组空间一般都是预分配的，不会频繁申请和释放，所以可以提供程序性能，这个做内存池优化的实现手段；

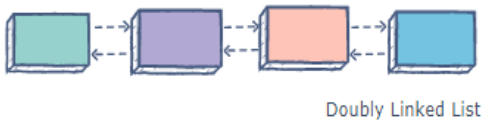
理解数据结构-链表

链表

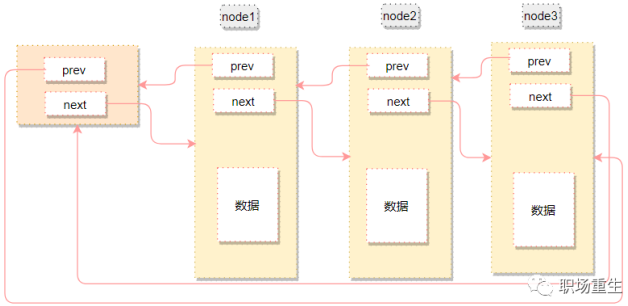
核心点：

- 按需分配，动态扩缩容，比较节约空间；

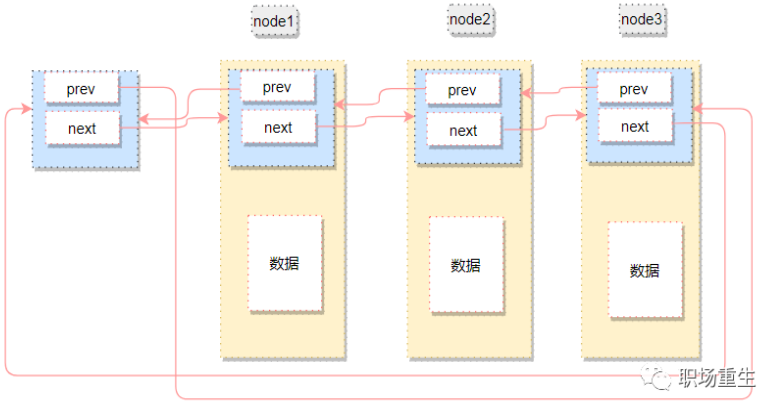




- 链表编程边界case检查：
  - 链表为空
  - 只包含一个节点
  - 只包含两个节点
  - 处理头节点和尾结点
- 每个节点必须有个“指针“要么指向其他节点，要么为空，这样才能把链表串起来，任何操作都必须保证链表完整性，不允许节点无故脱链，所以任何操作之前，都要思考会不会导致节点脱链，如果不下心脱链就会存在内存泄漏风险；
- 链表作为最**基础数据结构**，很多高级结构：队列，栈，hash，二叉树，都是在链表基础上演化而来；



- 编程技巧：
  - 链表最常规操作： CRUD，头结点处理，**快慢指针，**dummy node
  - 通常链表有两种实现方式，一种是抽象独立型，一种是传统耦合型
  - 链表最核心技巧，就是理解指针操作（包括安全检查-空指针判断），不要被指针复杂的赋值操作搞晕，多敲代码，找到经典的链表练习题（28原则）不断练习，比如：判断链表是否有环，反转链表，合并链表等，写好每一题，再认真总结，唯有熟才能生巧。

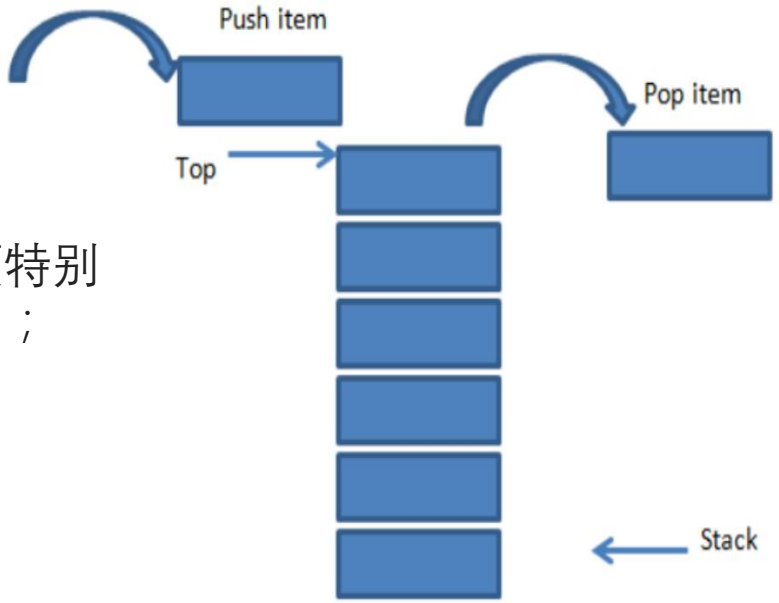


理解数据结构-**栈**

栈

核心点：

- 需要关注栈的深度大小（容量）；
- 栈各个极端情况处理，比如空，满，溢出等；
- 多线程下实现安全的栈操作；
- 堆栈很容易被攻击(缓冲区溢出攻击)，程序员必须特别注意避免这些实现的陷阱，防止代码产生安全漏洞；



使用场景：

- 操作系统程序运行栈，实现函数调用运行机制；
- 操作前进和后退，比如编辑器，浏览器等；
- 编译器使用，比如表达式解析，语法检查等；

理解数据结构-队列

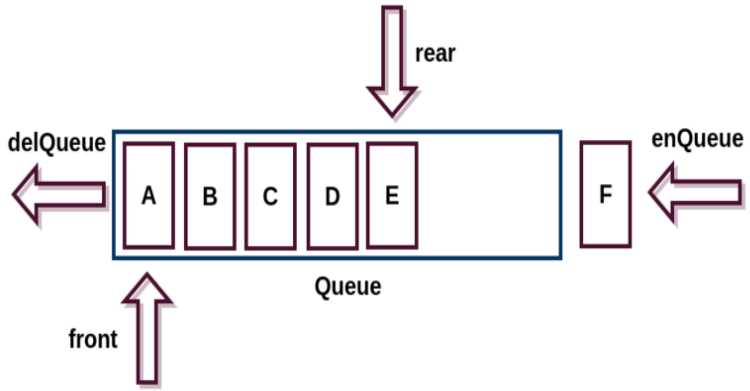
队列

核心点：

- 队列核心作用：应用耦合、异步处理、流量削锋（缓冲）；
- 队列各个极端情况处理，比如空，满，溢出等；
- 队列支持多线程并发操作，加锁或者**无锁队列**实现；
- 队列**零拷贝优化**，比如队列操作零拷贝（操作指针地址或者索引），IO零拷贝；

使用场景：

- 各种消息队列中间件，比如RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ、MetaMq等；
- 各种排队（缓冲区）系统，操作系统任务FIFO调度队列，数据包发送队列，凡是需要满足FIFO场景，都是可以用队列实现；

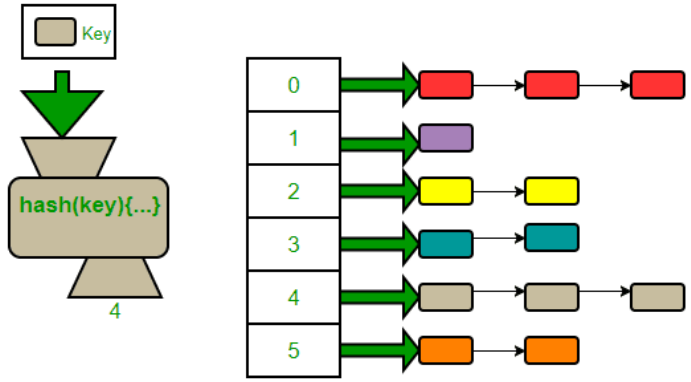


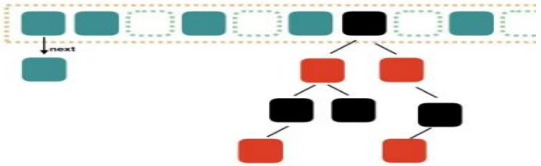
理解数据结构-Hash

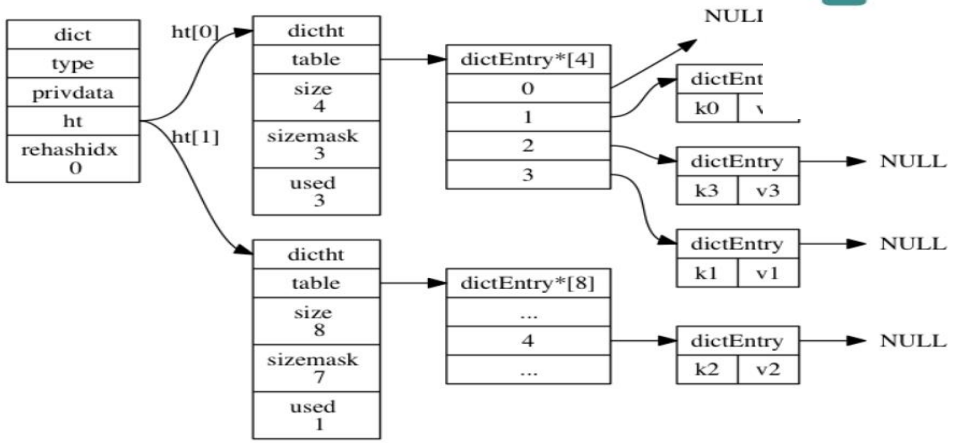
Hash

核心点：

- Hash是以空间换时间结构，需要评估好所有Hash头结构的内存使用量；
- Hash桶选择需要考虑到内存和冲突链长度大小（影响查找效率）；
- hash的优化--解决hash冲突
  - 内存有限场景下，就需要优化冲突链结构（链表转红黑树）；
  - 优化hash函数，让hash结果更均匀；
  - 可以考虑双重hash（空间和时间，还有编码复杂度的一种折中方案）
- 需要考虑hash扩容
  - rehash算法
  - 二次哈希
  - 一致性hash算法
  - 渐进式rehash算法
  - Linear Hash Tables
- 使用场景：
  - redis的dict结构设计；
  - 签名算法MD5算法；
  - 数据包校验 CRC算法；
  - 负载均衡LB选择后端服务器hash算法；
  - 布隆过滤器





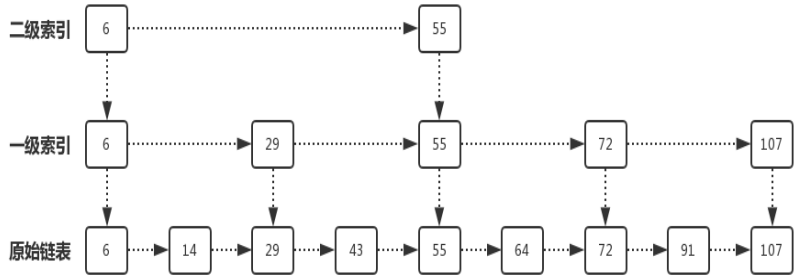


理解数据结构-**跳表(skip list)**

跳表

核心点：

- 跳表本质上是对链表的一种优化，通过逐层跳步采样的方式构建索引，以加快查找速度。使用概率均衡的思路，确定新插入节点的层数，使其满足集合分布，在保证相似的查找效率简化了插入实现。
- skiplist的复杂度和红黑树一样，而且实现起来更简单；
- 在并发环境下skiplist有另外一个优势，红黑树在插入和删除的时候可能需要做一些rebalance的操作，这样的操作可能会涉及到整个树的其他部分，而skiplist的操作显然更加局部性一些，锁需要盯住的节点更少，因此在这样的情况下性能好一些；



使用场景：

- HBase MemStore 的数据结构：HBase 属于 LSM Tree 结构的数据库，LSM Tree 结构的数据库有个特点，实时写入的数据先写入到内存，内存达到阈值往磁盘 flush的时候，会生成类似于 StoreFile 的**有序文件**，而跳表恰好就是天然有序的，所以在 flush 的时候效率很高。
- Google 开源的 key/value 存储引擎 LevelDB 以及 Facebook 基于 LevelDB 优化的 RocksDB 都是 LSM Tree 结构的数据库，他们内部的 MemTable 都是使用了跳表这种数据结构；
- redis的sorted set内部实现；

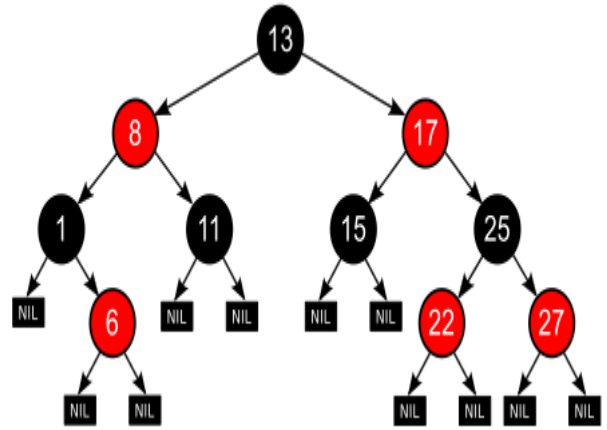
By 公众号@极客重生

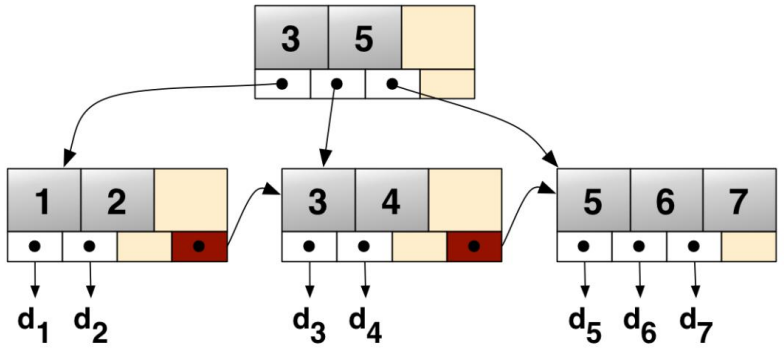
理解数据结构-树

树

核心点：

- 二叉查找树
- 平衡二叉树
  - AVL
  - 红黑树
  - …
- 多叉树（IO存储）：
  - Btree
  - B+tree
  - …
- 字典树-Tries树
- 区间树-线段树





理解数据结构-红黑树

红黑树

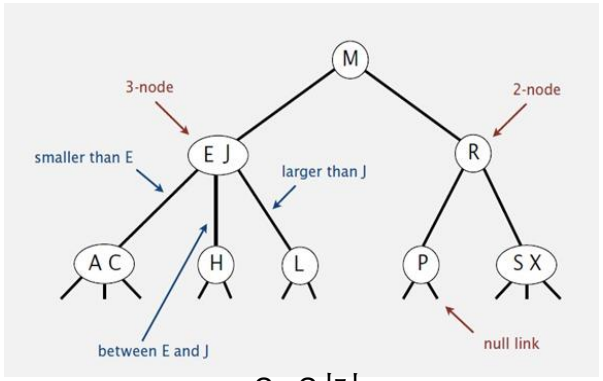
核心点

- 为什么还需要引入红黑树？我们希望数据结构具有关联性，即相邻版本之间，比如说第一次插入，和第二次插入时，树的结构不能发生太大变化，应该可以经过O（1）次数就可以变化完成。对于AVL树来说，插入是满足这个条件的，删除却不满足这个条件。红黑树就满足这一特性，插入和删除操作后的拓扑变化不会超过O(1)。
- 红黑树相对于AVL树来说，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。
- 维护红黑树的平衡需要考虑7种不同的情况。
- 红黑树为什么综合性能好？
  - 2-3树
  - 缓存利用率高
  - 减少旋转次数，使再平衡尽快结束。

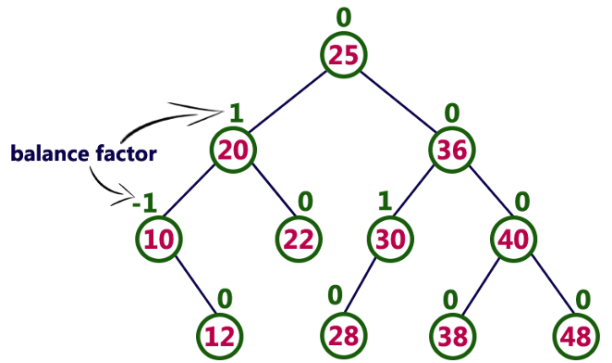
By 公众号@极客重生

使用场景（更新）：

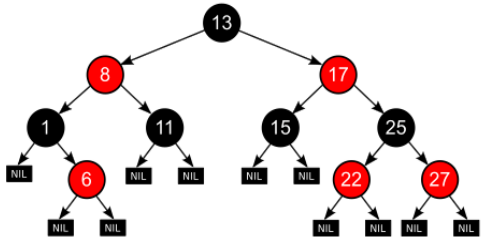
- 广泛用在C++的STL中。如map和set都是用红黑树实现的；
- Java TreeMap和TreeSet 都是基于红黑树实现的，而 JDK8 中 HashMap 当链表长度大于 8 时也会转化为红黑树
- CFS进程调度算法中，vruntime利用红黑树来进行存储，选择最小vruntime节点调度。
- 定时器使用rbtree来组织未完成的计时器请求。
- ext3文件系统跟踪红黑树中的目录条目。
- 虚拟内存结构管理（VMA)。
- 多路复用技术的Epoll的核心结构也是红黑树+双向链表。
- 加密密钥和网络数据包均由红黑树跟踪。



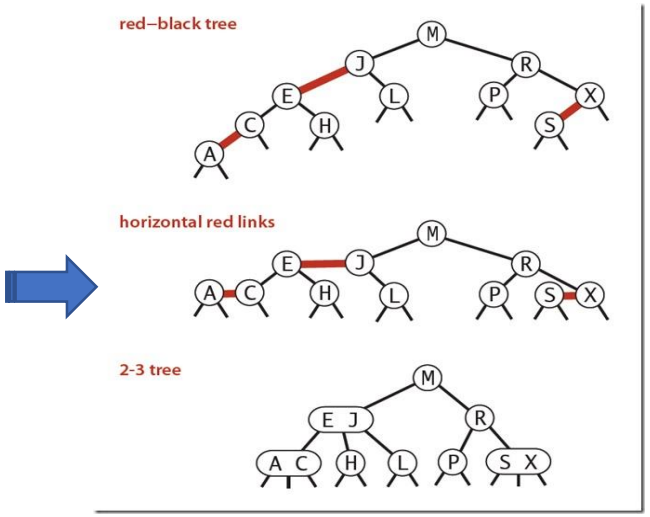
2-3树



AVL



红黑树

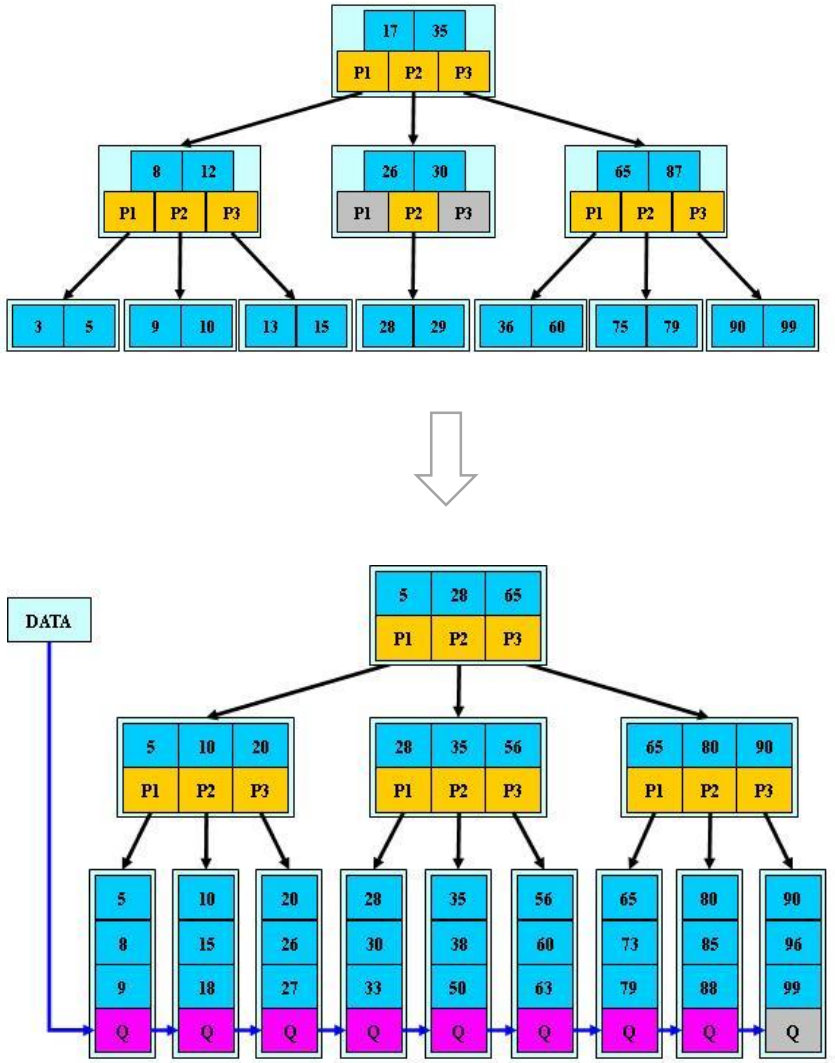


理解数据结构-BTree

BTree

核心点：

- 出现背景：B 树和 B ＋树的出现是因为磁盘IO，IO操作的效率很低，那么当在大量数据存储中，查询时我们不能一下子将所有数据加载到内存中，只能逐一加载磁盘页，每个磁盘页对应树的节点。造成大量磁盘ＩＯ操作（最坏情况下为树的高度）。平衡二叉树由于树深度过大而造成磁盘IO读写过于频繁，进而导致效率低下。所以，我们为了减少磁盘ＩＯ的次数，就你必须降低树的深度，将“瘦高”的树变得“矮胖”。
  - 每个节点存储多个元素
  - 摒弃二叉树结构，采用多叉树
- BTree是为磁盘等外存储设备设计的一种平衡查找树。系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，B+Tree是在B-Tree基础上 的一种优化，使其更适合实现外存储索引结构；
- B+Tree 只有叶子结点存储数据，所有非叶子结点（内部结点）不存储数据，只有指向子结点的指针。叶子结点均在同一层，且叶子结点之间类似于链表结构，即有指针指向下一个叶子结点；
- 由于B+树在内部节点上不包含数据信息，因此在内存页中能够存放更多的key。数据存放的更加紧密，具有更好的空间局部性。因此访问叶子节点上关联的数据也具有更好的缓存命中率
- B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历，相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好，B树也有优点，其优点在于：由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。



使用场景：

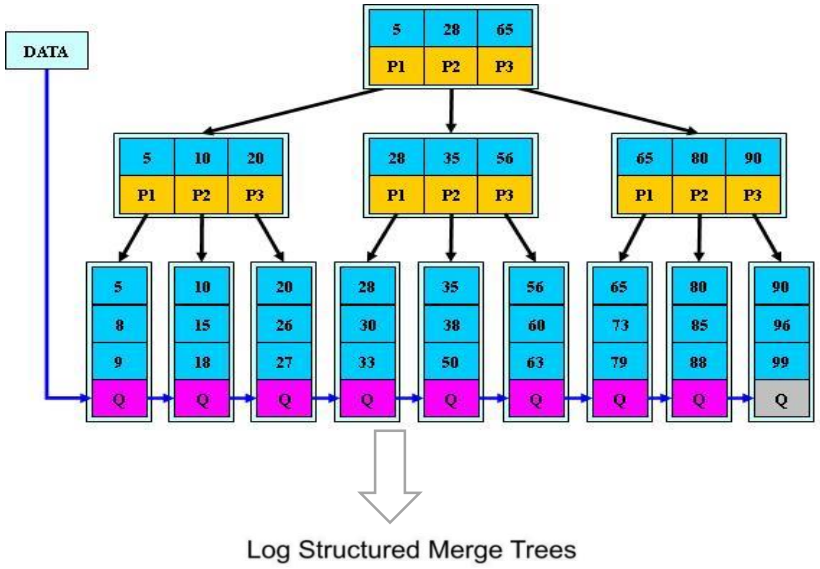
- 数据库存储引擎：MySQL InnoDB存储引擎就是用B+Tree实现：
- Rust的treeMap容器。

理解数据结构-树对比

树对比

BST（二叉搜索树） vs BTree（多叉）

- BST确实是理论上内存数据结构的最优解，但是有个前提：内存是真的均质随机访问内存。这里给出一个定义，均质随机访问内存即主存拥有在任意上下文场景下，访问任意地址都有着非常相似的性能。在计算机当中，由于cache的存在，访问临近位置的内存在平均意义下会产生非常巨大的性能提升，而BST的特性导致临近的元素并不是在内存中存放在一起的，从而在实践当中性能非常糟糕。而BTree在大部分场景下，可以让一些临近元素在内存中存放在一起，从而在大部分情况下，实践中得到比BST更好的性能。



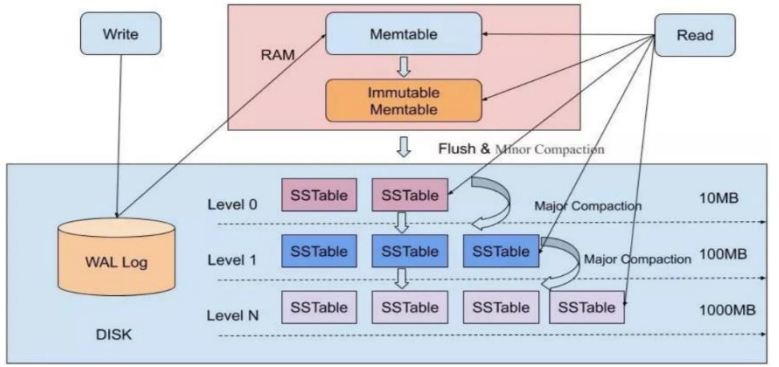
Log Structured Merge Trees

BTree 升级

- **B树：**多路搜索树，每个结点存储M/2到M个关键字，非叶子结点存储指向关键字范围的子结点；所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；
- **B+树：**在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；
- **B*树：**在B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从1/2提高到2/3。

BTree vs LSM Tree

- LSM树（Log Structured Merge Tree）设计思路是将数据拆分为几百M大小的Segments，并是顺序写入。将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘（由此提升了写性能，是一种基于硬盘的数据结构，与BTree相比，能显著地减少硬盘磁盘臂的开销。当然凡事有利有弊，LSM树和B+树相比，LSM树牺牲了部分读性能，用来大幅提高写性能，适合大量数据写入场景（比如各种NoSQL在大数据领域的应用）。



数据连续性对比：

- BST(几乎完全不连续) < BTree(部分连续) < LSMT（几乎全连续，偶尔不连续）

理解算法的本质

算法

- 算法是基于某种数据结构，快速高效的实现数据操作的方法。
- 部分算法问题本质是数学的问题，只是通过计算机这个工具来解决数学计算问题

宏观上看，所有计算机问题都可以用穷举来解决，穷举所有可能性。算法设计就是先思考“如何穷举”，然后再追求“如何聪明地穷举”(性能优化，空间和时间)

理解算法的本质

By 公众号@极客重生

排序

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

每种排序算法都各有优缺点。因此，在实用时需根据不同情况适当选用，甚至可以将多种方法结合起来使用。

核心点：

分类：

- 插入排序法：直接插入排序、希尔排序
- 交换排序法：冒泡、快排
- 选择排序法：选择排序、堆排序
- 归并排序：归并排序
- 非比较排序：计数排序，桶排序，基数排序

时间复杂度:

- 平方阶(O(n2))排序
  - 各类简单排序:直接插入、直接选择和冒泡排序；
- 线性对数阶(O(nlog2n))排序
  - 快速排序、堆排序和归并排序；
- 线性阶(O(n))排序
  - 基数排序，此外还有桶、箱排序。

实际选择：

1）当n较大，则应采用时间复杂度为O(nlog2n)的排序方法：

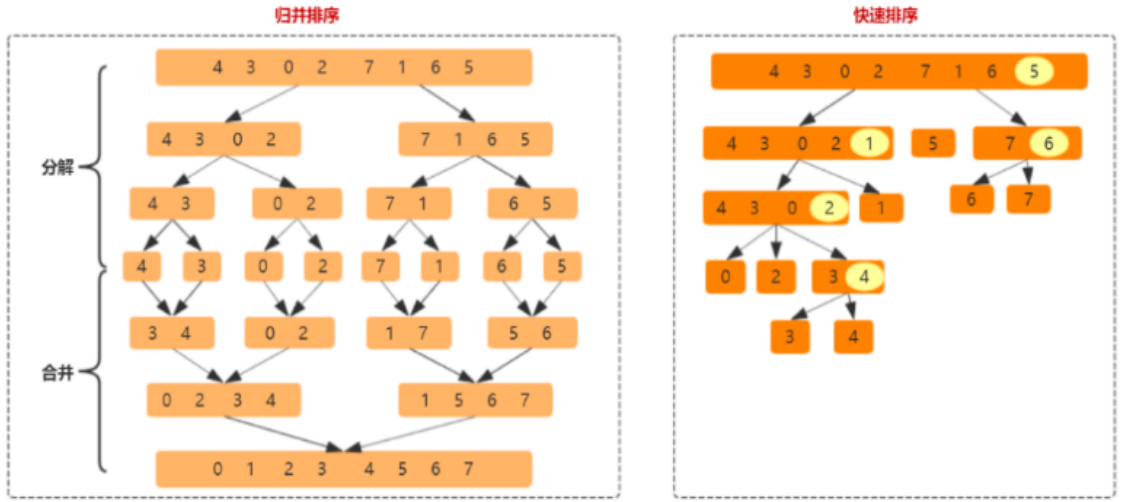
快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

2） 当n较大，内存空间允许，且要求稳定性 =》**归并排序**

3）当n较小，可采用**插入排序**或选择排序。

直接插入排序：当元素分布有序，直接插入排序将大大减少比较次数和移动记录的次数。

4）冒泡很少使用，基数排序它是一种稳定的排序算法，但有一定的局限性。



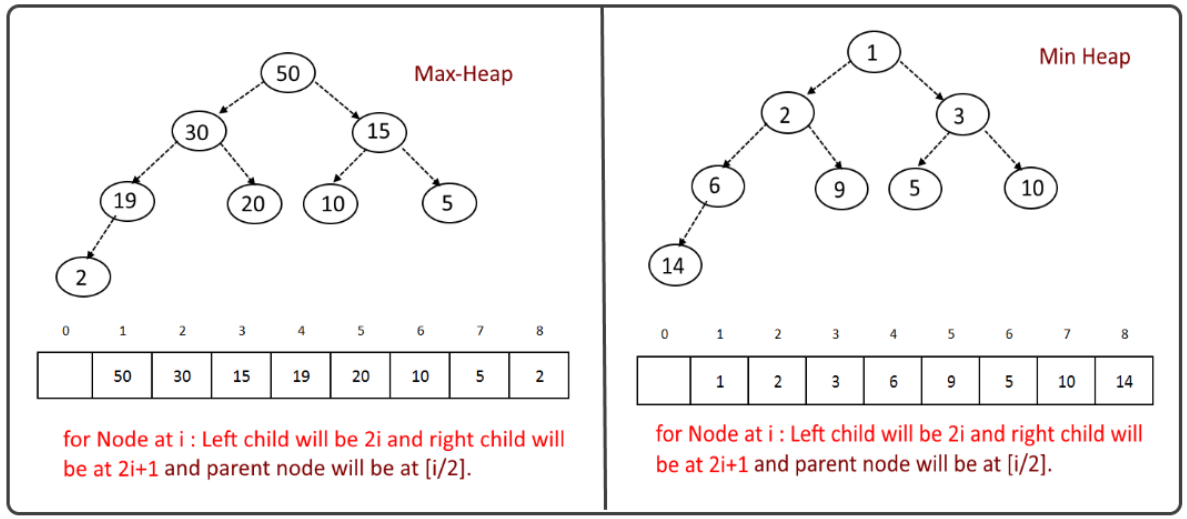
数据规模	快速排序	归并排序	希尔排序	堆排序
1000万	0.75	1.22	1.77	3.57
5000万	3.78	6.29	9.48	26.54
1亿	7.65	13.06	18.79	61.31

By 公众号@极客重生

什么场景下归并排序比快排高效？

为什么通常情况下快排比堆排序要高效？

快速排序的最直接竞争者是堆积排序（Heapsort）



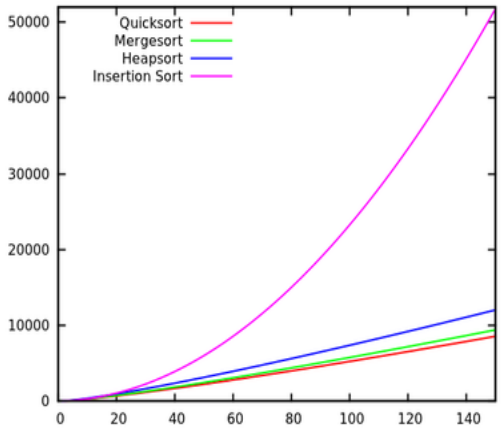
- 堆排序访问数据的方式没有快速排序友好，快排的局部性更友好。
- 对于同样的数据，在排序过程中，在堆排里面有大量这种近乎无效的比较，堆排序算法的数据交换次数要多于快速排序。

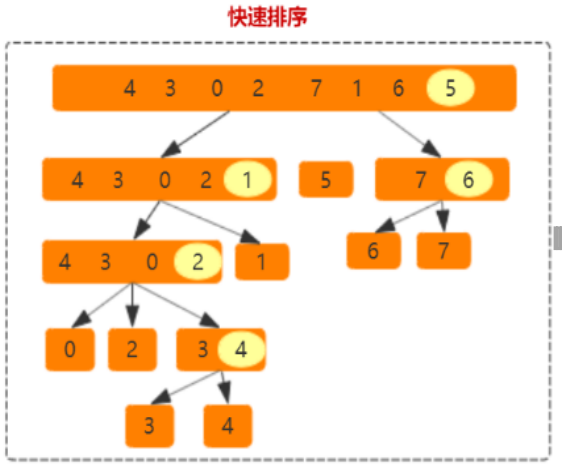
所有基于比较的排序都逃脱不了NlogN的宿命，为什么？

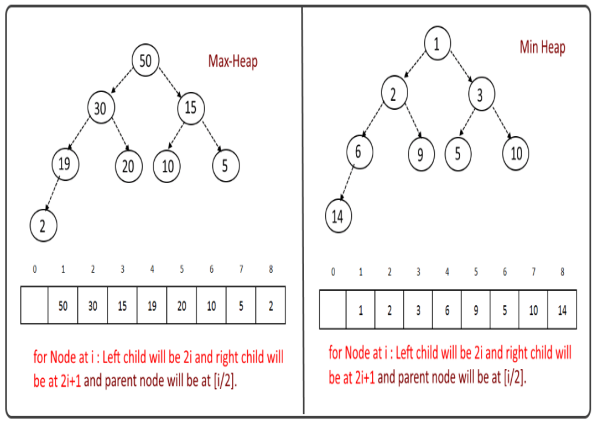
标准库中的sort实现

它以quicksort开始，当递归深度超过基于被排序元素数量（的对数）的级别时，它切换到堆排序，当元素数量低于某个阈值时，它切换到插入排序。

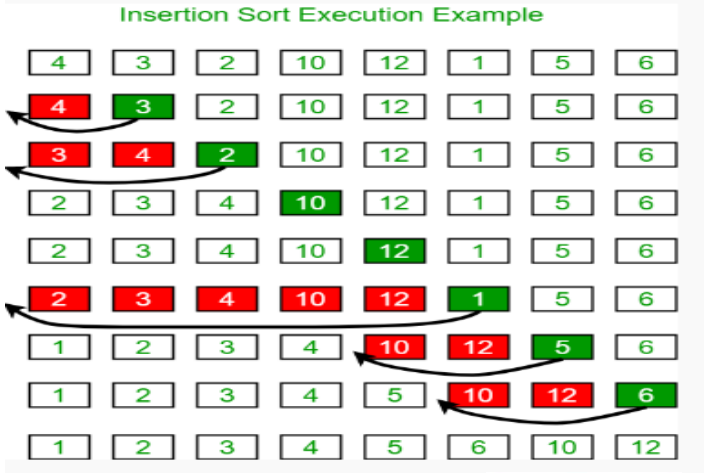
插入排序：
(+) 小输入最快
(-) 大多数大输入的二次方
快速排序：
(+) 大多数输入都快
(+) 缓存友好
(-) 最坏情况N^2
堆排序：
(+) 保证 O(nlogn)
(+) 就地工作,即使用 O(1) 额外内存
(-) 几乎总是在nlogn中运行，即使输入已排序
(-) 实际性能比QuickSort差







From The Art of Computer Programming 三个排序算法复杂度：
快速排序: 11.667(n+1)ln(n)−1.74n−18.74
堆排序: 16nln(n)+0.01n
插入排序: 2.25n2+7.75n−3ln(n)



- 2000 年 6 月的SGI C++标准模板库 stl_algo.h实现的不稳定排序使用 Musser introsort 方法，递归深度切换到作为参数传递的堆排序，3 中值枢轴选择和 Knuth 最终插入排序传递较小的分区超过 16。
- GNU 标准 C++ 库类似：使用最大深度为 2×log 2 n的introsort，然后对小于 16 的分区进行插入排序。 [3]
- LLVM libc++还使用最大深度为 2×log 2 n的 introsort，但是对于不同的数据类型，插入排序的大小限制是不同的（如果交换很简单，则为 30，否则为 6）。此外，大小最大为 5 的数组将单独处理。[4]
- Go使用经过少量修改的 introsort：对于 12 个或更少元素的切片，它使用Shellsort而不是插入排序，并且更高级的中位数为三个枢轴选择的三个中位数用于快速排序。
- Java从版本 14 (2020) 开始使用混合排序算法，该算法对高度结构化的数组（由少量已排序子数组组成的数组）使用归并排序，否则使用 introsort 对整数、长整数、浮点数和双精度数组进行排序. [6]

排序算法的未来：introspective sort （自适应排序算法）根据不同场景选择结合了多种算法的优点于一身，算法具有快速的平均性能和最佳的最坏情况性能

理解算法的本质

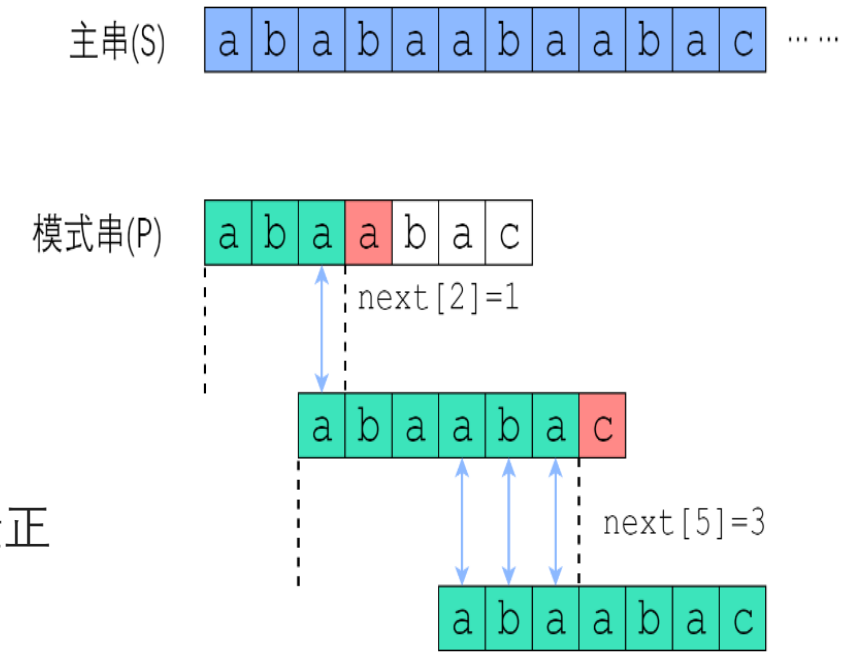
字符串匹配算法

核心点：

- 尽可能利用一切可以利用信息，比如模式串本身信息，后缀信息，比较后的残余信息等；
- 掌握正则表达式语法；
- 理解模式匹配：KMP、Boyer-Moore算法；

使用场景：

- linux文本处理三剑客 grep ， awk， sed 等用了大量正则表达式算法。
- 文本编辑器使用大量字符串匹配算法。



信息越多，提高剪枝效率，排除无用操作就会越多，提升性能

理解算法的本质

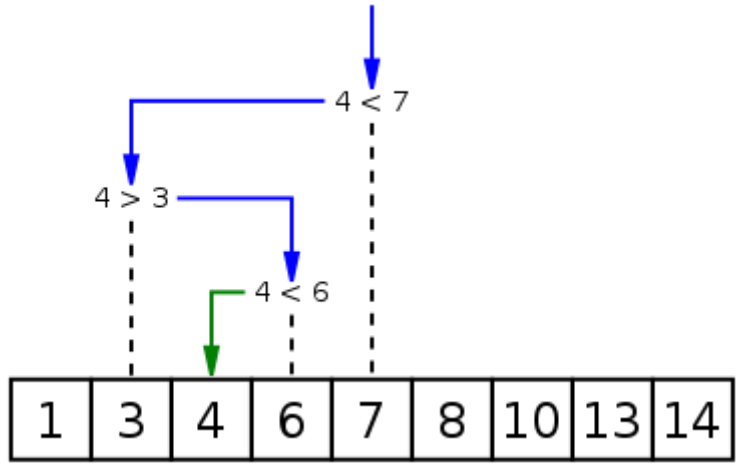
二分法

核心点：

- 前置条件，需要有序；
- 全面考虑算法边界点，实现细节，溢出等问题；
- 不能在线增量计算；

使用场景：

- 用二分法计算方程近似解；
- 机器学习二分分类算法；
- 有序序列快速查找场景；




```
int binary_search(const int arr[], int start, int end, int key) {
    int ret = -1;       // 未搜索到数据返回-1下标

    int mid;
    while (start <= end) {
        mid = start + (end - start) / 2; //直接平均可能會溢位，所以用此算法
        if (arr[mid] < key)
            start = mid + 1;
        else if (arr[mid] > key)
            end = mid - 1;
        else {            // 最後檢測相等是因為多數搜尋狀況不是大於要不就小於
            ret = mid;
            break;
        }
    }

    return ret;     // 单一出口
}
```

猜数字游戏，你会怎么玩？

二分法保证了最坏情况下性能底线，“最好的问题”就是那些能够均分所有可能性的问题

理解算法的本质

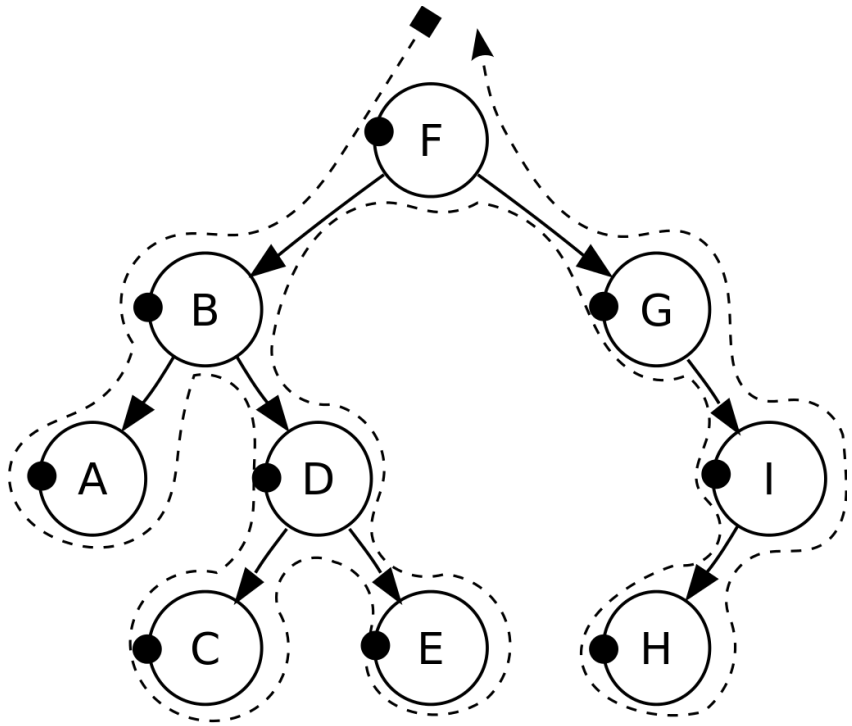
搜索(遍历)+剪枝

核心点：

- 核心是减少解空间，穷举+排除法；
- 核心是减少解空间，递归+判断；

使用场景：

- 深度优先搜索
- 广度优先搜索
- A*算法、回溯算法、蒙特卡洛树搜索；



理解算法的本质

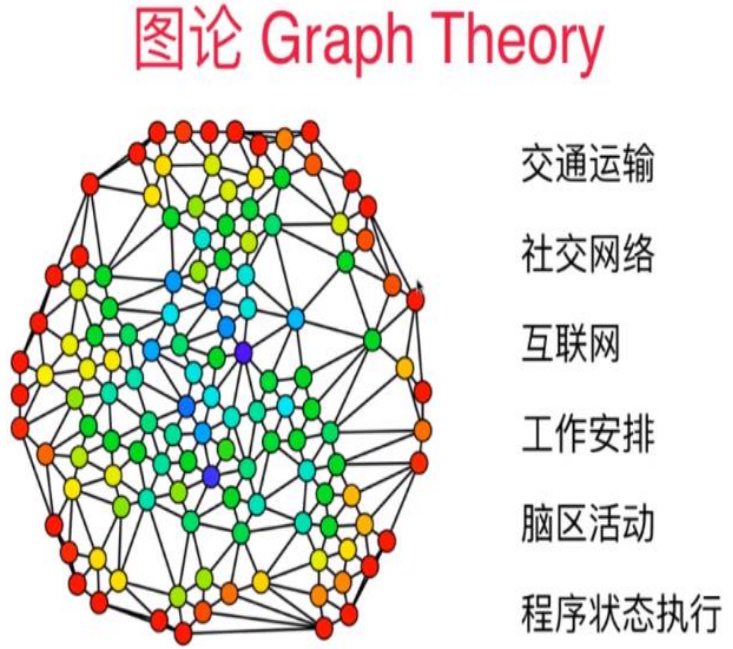
图算法

核心点：

- 实际工作中网络会用到部分图算法，比如网络拓扑排序，OSPF路由协议（最短路径）；
- 如果你要参加ACM or OJ 这种比赛，可以学一些常用图算法：最短路径、最小生成树、网络流建模；

使用场景：

- 游戏开发中会用到大量的图算法：路径搜索算法（BFS,DFS,A*）；
- 导航软件会用到大量的图算法：最短路径，路径搜索算法；
- 网络bridge的STP协议用到最小生成树算法；
- 网页排名中PageRank 算法；



理解算法的本质

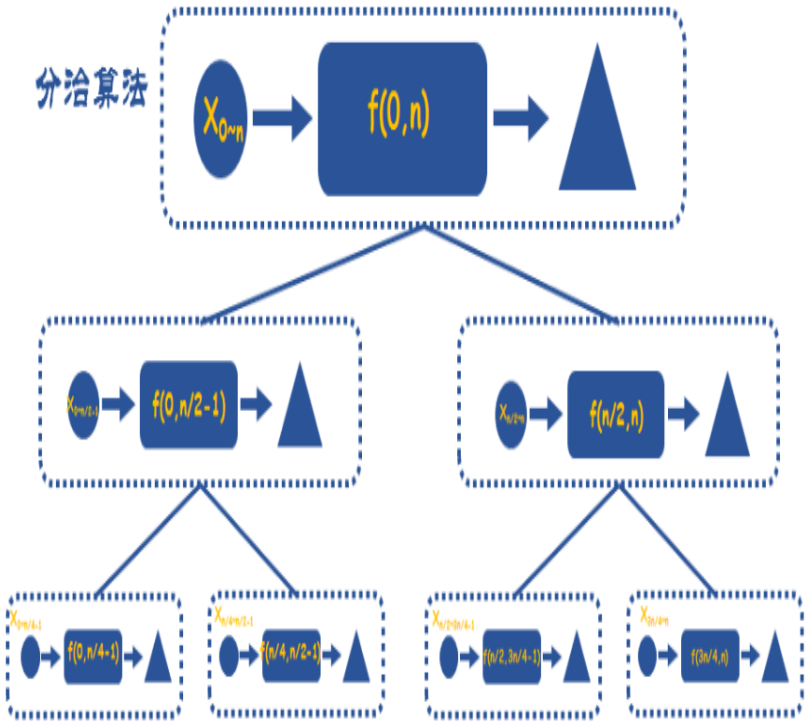
分治算法

核心点：

- 一是自顶向下分解问题，二是自底向上抽象合并；
- 将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之；

使用场景：

- 递归，二分搜索，排序算法(快速排序，归并排序)，动态规划；
- 傅立叶变换(快速傅立叶变换)；
- 程序模块化设计；



理解算法的本质

动态规划
(DP)

核心点: 实际工作中很少用到，工作多年，从来没有在工作中使用过动态规划，核心是了解其思想和原理.

动态规划最核心的思想：

- 求解动态规划的核心问题是穷举。因为要求最值，肯定要把所有可行的答案穷举出来，然后在其中找最值
- 动态规划的穷举有点特别，因为这类问题存在「重叠子问题」，如果暴力穷举的话效率会极其低下，所以需要备忘录或者DP table（状态表）来优化穷举过程，避免不必要的计算
- 动态规划问题一定会具备「最优子结构」，才能通过子问题的最值得到原问题的最值。

编程重点：明确状态变量, 定义 dp数组(函数)的含义，写出DP状态转移方程，以及考虑边界条件；
面试：刷几道最常规DP题就行，一般面试很少出动态规划的题，校招可能会出，社招基本上不会出；
比赛：如果你要参加ACM or OJ 这种比赛，必须掌握, 这个区分菜鸟的分界点。熟悉各种类型DP特点，普通DP，区间DP，树形DP，数位DP，状态压缩DP等；

选择：一个问题是该用递推、贪心、搜索还是动态规划，完全是由这个问题本身阶段间状态的转移方式决定的！
每个阶段只有一个状态->**递推**；
每个阶段的最优状态都是由上一个阶段的最优状态得到的->**贪心**；
每个阶段的最优状态是由之前所有阶段的状态的组合得到的->**搜索**；
每个阶段的最优状态可以从之前
某个阶段的某个或某些状态直接得到而不管之前这个状态是如何得到的->**动态规划。**

使用场景：
1 切割钢条问题, Floyd最短路问题,最大不下降子序列,矩阵链乘,凸多边形三角剖分,0-1背包,最长公共子序列,最优二分搜索树；
2 运筹学，经济学等；

最长公共子序列

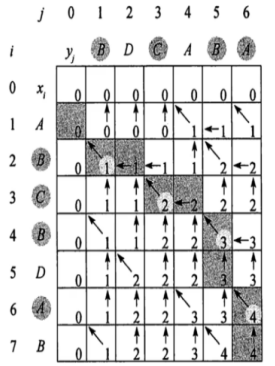
定理 15.1(LCS的最优子结构) 令 $X=\langle x_1, x_2, \cdots, x_m\rangle$和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$为两个序列，$Z=\langle z_1, z_2, \cdots, z_k\rangle$为 X 和 Y 的任意 LCS。

1. 如果 $x_m=y_n$，则 $z_k=x_m=y_n$ 且 Z_{k-1}是 X_{m-1}和 Y_{n-1}的一个 LCS。
2. 如果 $x_m\neq y_n$，那么 $z_k\neq x_m$ 意味着 Z 是 X_{m-1}和 Y 的一个 LCS。
3. 如果 $x_m\neq y_n$，那么 $z_k\neq y_n$ 意味着 Z 是 X 和 Y_{n-1}的一个 LCS。

最优解的递推式：
$$c[i,j]=\begin{cases}0 & \text{若 } i=0 \text{ 或 } j=0\\ c[i-1,j-1]+1 & \text{若 } i,j>0 \text{ 且 } x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{若 } i,j>0 \text{ 且 } x_i\neq y_j\end{cases}$$

c[i,j]表示X_i和Y_j的LCS长度

最长公共子序列



$$c[i,j]=\begin{cases}0 & \text{若 } i=0 \text{ 或 } j=0\\ c[i-1,j-1]+1 & \text{若 } i,j>0 \text{ 且 } x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{若 } i,j>0 \text{ 且 } x_i\neq y_j\end{cases}$$

By 公众号@极客重生

总结

总结

程序 = 数据结构 + 算法

程序本质是数据结构+算法，任何一门语言都可以这样理解，这个公式对计算机科学的影响程度足以类似物理学中爱因斯坦的“E=MC^2”——一个公式展示出了**程序的本质**。

工作中，最重要是设计好数据结构，因为在设计数据结构的时候，就是**性能和实现难度的权衡**，程序当然是越简单越好，但为了性能，有时候不得不设计出像红黑树这种复杂数据结构

希望我们都可以掌握好**算法核心思想**，和常见**数据结构精妙设计**，帮助我们在面试和工作中打好坚实的基础；

算法与数据结构-训练任务

Alex | 大师兄

2022-11-07 22:55

第四章-算法与数据结构-训练任务

基础：实现一遍基础数据结构（CURD）和算法（链表，队列，栈，二叉树，插入排序，快排，堆排序，二分查找，深度和广度遍历，分治算法（递归），最长子序列，最小生成树，最短路径）

训练比赛：精选几道算法训练题（链表，栈和队列，hash，树，二分，快排，堆等）进行一轮小比赛（一周，难度不高，都是常见的面试题），看谁的代码更好，更快，bug更少。

1 反转链表：https://link.zhihu.com/?target=https%3A//leetcode-...

2 合并K个有序链表：https://leetcode.cn/problems/merge-k-sorted-lists/

3 栈和队列：https://leetcode.cn/problems/evaluate-reverse-poli...

4 堆：https://leetcode.cn/problems/top-k-frequent-elemen...

5 hash：https://link.zhihu.com/?target=https%3A//leetcode-...

6 二分：https://leetcode.cn/problems/find-peak-element/

7 排序：https://leetcode.cn/problems/largest-number/descri...

8 树：https://leetcode.cn/problems/kth-smallest-element-...

9 二叉树的最近公共祖先：https://leetcode.cn/problems/lowest-common-ancesto...

10 动态规划DP：https://leetcode.cn/problems/longest-consecutive-s...

进阶：分析标准库sort源码

收起