**算法大全**

## 一、Tree

**1 内容摘要**

B-Tree 介绍

B-Tree 特性搜索插入等

B+Tree 介绍

B\*Tree 介绍

**2 B-Tree 介绍**

1970年，R.Bayer和E.mccreight提出了一种适用于外查找的树，它是一种平衡的多叉树，称为B树，其定义如下。一棵m阶的B树满足下列条件：

@树中每个结点至多有m个孩子；

@除根结点和叶子结点外，其它每个结点至少有m/2个孩子；

@若根结点不是叶子结点，则至少有2个孩子；

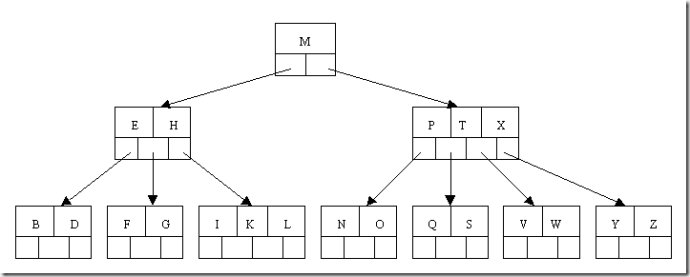
@所有叶子结点(失败节点)都出现在同一层，叶子结点不包含任何关键字信息；

@所有非终端结点中包含下列信息数据 ( n, A0 , K1 , A1 , K2 , A2 , … , Kn , An )，其中： Ki (i=1,…,n)为关键字，且Ki < Ki+1 , Ai (i=0,…,n)为指向子树根结点的指针, n为关键字的个数

@非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；

在B树中，每个结点中关键字从小到大排列，并且当该结点的孩子是非叶子结点时，该k-1个关键字正好是k个孩子包含的关键字的值域的分划。因为叶子结点不包含关键字，所以可以把叶子结点看成在树里实际上并不存在外部结点，指向这些外部结点的指针为空，叶子结点的数目正好等于树中所包含的关键字总个数加1。B树中的一个包含n个关键字，n+1个指针的结点的一般形式为： （n,P0,K1,P1,K2,P2,…,Kn,Pn）其中，Ki为关键字，K1 <K2 <… <Kn, Pi 是指向包括Ki到Ki+1之间的关键字的子树的指针。

-- 图1 B-tree示意图



**3 B-Tree特性**

**（1） B-Tree 特性**

@关键字集合分布在整颗树中；

@任何一个关键字出现且只出现在一个结点中；

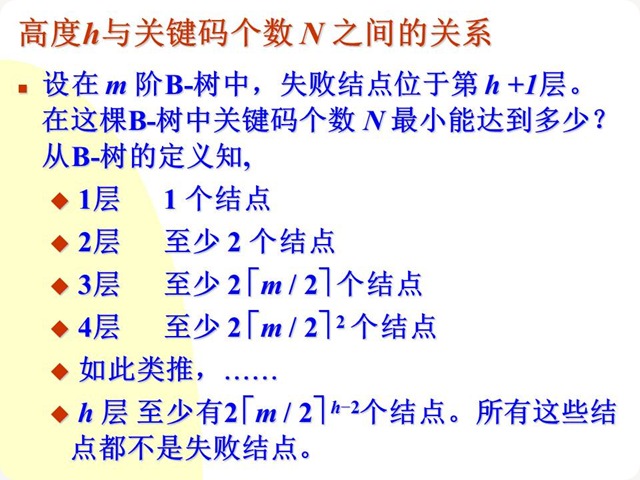
@搜索有可能在非叶子结点结束；

@其搜索性能等价于在关键字全集内做一次二分查找；

@自动层次控制；

**（2） B-Tree搜索原理**

B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；因此，B-Tree的查找过程是一个顺指针查找结点和在结点的关键字中进行查找的交叉进行的过程。



-- 图2 高度与关键码的计算过程

**（3）B-Tree 插入**

B-树是从空树起，逐个插入关键码而生成的。

在B-树，每个非失败结点的关键码个数都在[ m/2 -1, m-1]之间。插入在某个叶结点开始。如果在关键码插入后结点中的关键码个数超出了上界 m-1，则结点需要“分裂”，否则可以直接插入。

实现结点“分裂”的原则是：

设结点 A 中已经有 m-1 个关键码，当再插入一个关键码后结点中的状态为( m, A0, K1, A1, K2, A2, ……, Km, Am)其中 Ki < Ki+1, 1 =< m

这时必须把结点 p 分裂成两个结点 p 和 q，它们包含的信息分别为：

结点 p：( m/2 -1, A0, K1, A1, ……, Km/2 -1, Am/2 -1)

结点 q：(m - m/2, Am/2, Km/2+1, Am/2+1, ……, Km, Am)

位于中间的关键码 Km/2 与指向新结点 q 的指针形成一个二元组 ( Km/2, q )，插入到这两个结点的双亲结点中去。

**4 B+Tree**

**（1） B+Tree定义**

B+树可以看作是B树的一种变形，在实现文件索引结构方面比B树使用得更普遍。

一棵 m 阶B+树可以定义如下：

树中每个非叶结点最多有 m 棵子树；

根结点 (非叶结点) 至少有 2 棵子树。除根结点外, 其它的非叶结点至少有 ém/2ù 棵子树；有 n 棵子树的非叶结点有 n-1 个关键码。

所有叶结点都处于同一层次上，包含了全部关键码及指向相应数据对象存放地址的指针，且叶结点本身按关键码从小到大顺序链接；

每个叶结点中的子树棵数 n 可以多于 m，可以少于 m，视关键码字节数及对象地址指针字节数而定。

若设结点可容纳最大关键码数为 m1，则指向对象的地址指针也有 m1 个。

结点中的子树棵数 n 应满足 n 属于[m1/2, m1]

若根结点同时又是叶结点，则结点格式同叶结点。

所有的非叶结点可以看成是索引部分，结点中关键码 Ki 与指向子树的指针 Pi 构成对子树 (即下一层索引块) 的索引项 ( Ki, Pi )，Ki 是子树中最小的关键码。

特别地，子树指针 P0 所指子树上所有关键码均小于 K1。结点格式同B树。

叶结点中存放的是对实际数据对象的索引。

在B+树中有两个头指针：一个指向B+树的根结点，一个指向关键码最小的叶结点。

**5 B+Tree与B-Tree区别**

@非叶子结点的子树指针与关键字个数相同；

@非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

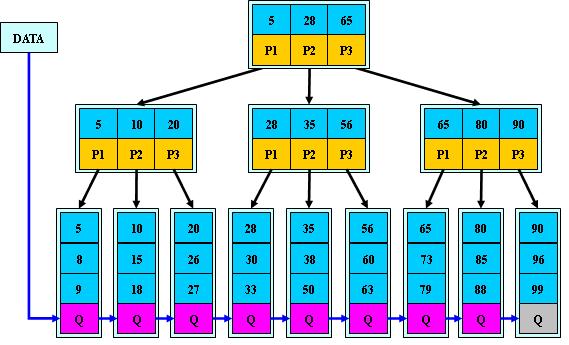
@为所有叶子结点增加一个链指针；

@所有关键字都在叶子结点出现；

对B+树进行两种搜索运算

循叶结点链顺序搜索

另一种是从根结点开始，进行自顶向下，直至叶结点的随机搜索。



-- 图3 B+Tree示意图

**6 B+Tree的特性**

@所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

@不可能在非叶子结点命中；

@非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

@更适合文件索引系统

**7 三者比较**

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；

## 二、Diagram

**1 基本算法**

①建立图的存储结构

②深度优先搜索和广度优先搜索

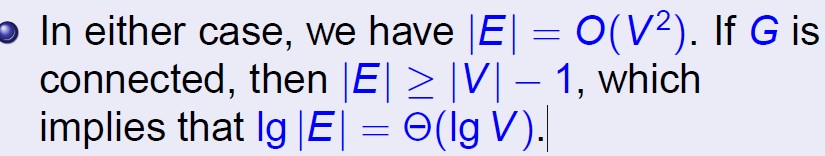
③求图的最小生成树

④拓扑排序

⑤最短路径

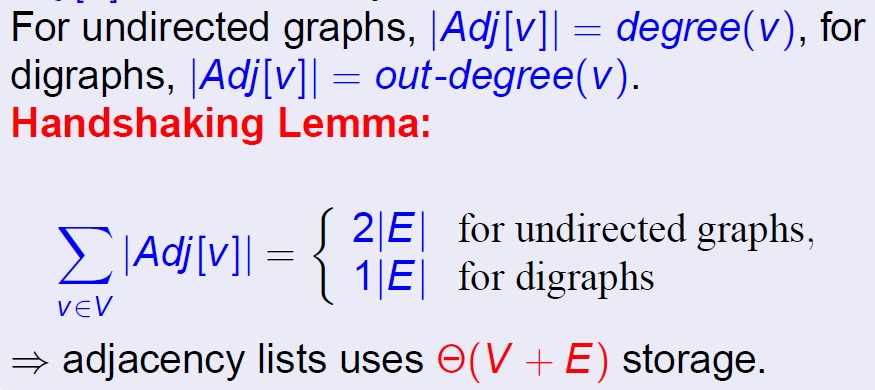
**2 图的分类**

图分为有向图和无向图，两种都可以表示为G=｛V,E｝，分别是顶点和边，有向无向均是针对边来说的。下面的lgE=lgV是同一个数量级的这个在最小生成树算法的算法复杂度分析中比较重要，所以这里罗列出来。



**3 图的表示方法有两种，邻接链表和邻接矩阵**

邻接链表有一个握手定理，就是遍历邻接表中所有点的邻接点，在无向图中需要2E次，有向图中需要E次。这里在图的算法分析中还是比较重要的。如下：



留下两个思考题：

针对邻接矩阵表示的图和邻接链表表示的图G，进行反转的时间复杂度分别是多少呢？

针对邻接矩阵，直接进行0转1，1转0即可，时间复杂度矩阵的大小；

但是邻接链表？我自己的思路是重新构造一个邻接链表，新的链表顶点，针对顶点1，把它加入到其后邻接点的作为顶点，依次类推，这个时间复杂度为E；

针对邻接矩阵表示的图和邻接链表表示的图G，将图中路径为长度为2的点进行连接这种图的变换的时间复杂度分别是多少呢？

针对邻接矩阵，同样比较简单，直接将矩阵平方即可，时间复杂度为矩阵乘法的时间复杂度；

邻接链表也只是我个人理解的一个思路：重新构造一个邻接链表，第一个遍历链表后面元素的链表...有点绕，但是就是顶点的邻接顶点作为顶点的邻接顶点，但是要有去重的工作，没有去重则为2E；

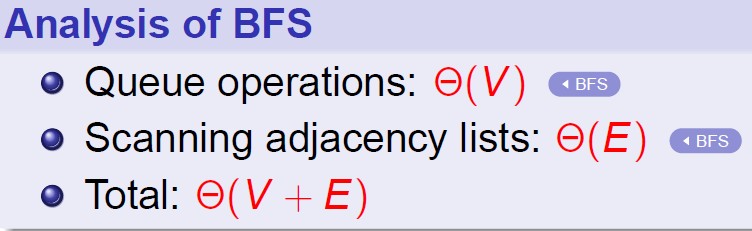
**4 广度优先搜索**

图的搜索算法都是根据标记颜色来进行的，接下来的深度优先搜索也是有颜色的标记。

伪代码：

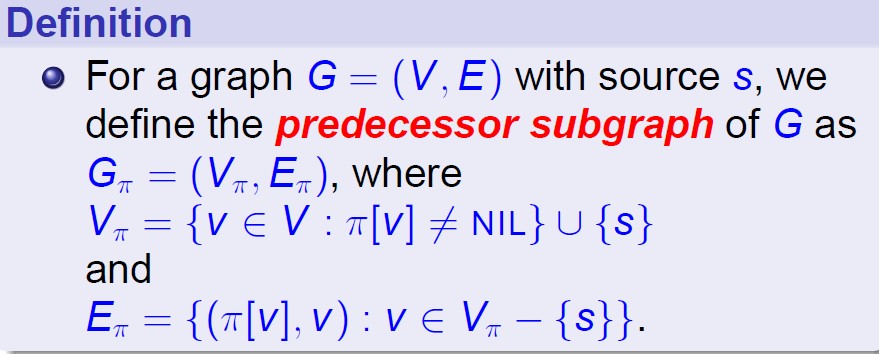
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | BFS(G,s)      for each vertex u in V[G]-s          color[u]=white          father[u]=NIL          depth[u]=MAX      color[s]=gray      depth[s]=0      father[s]=NIL      Q=NULL      ENQUEUE(Q,s)      while(Q!=NULL)          u=DEQUEUE(Q)          for each v in Adj[u]              if color[v]=white                  color[v]=gray                  depth[v]=depth[u]+1                  father[v]=u                  ENQEUE(Q,v)           color[u]=black |

时间复杂度分析



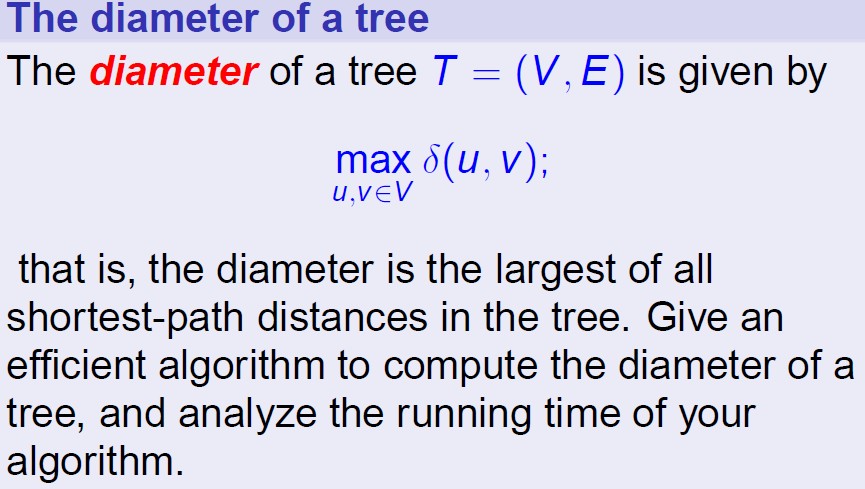
 广度优先树

广度优先搜索形成一个树的结构，其定义如下由上面形成的father结构形式



可以递归的输出广度优先树的到达开始节点的路径，这个路径也是最短路径。

BFS思考题：



思路一：一个点BFS找到一个最远的点，然后从这个最远的点开始BFS，到达的最远的一个点的距离及为直径

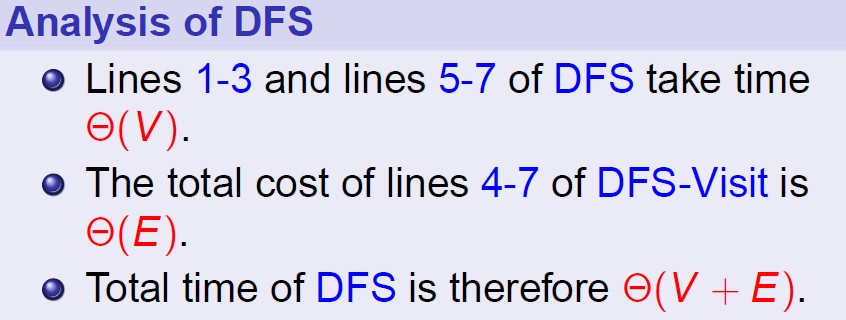
思路二：根据根节点分为左子树和右子树，然后左子树的高度+1+右子树的高度为直径？

**5 深度优先搜索**

深度优先搜索有一个时间戳的标识，伪代码：

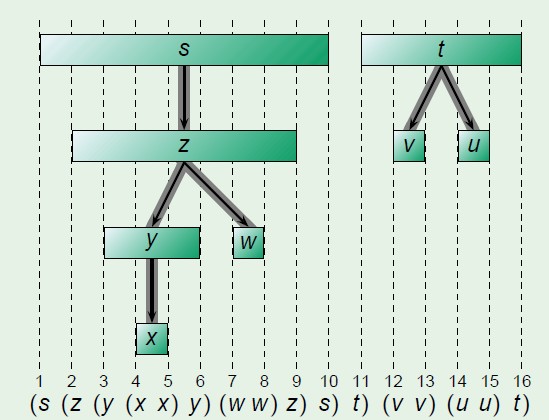
|  |
| --- |
| DFS(G)      for each u in V[G]          color[u]=white          father[u]=NIL      time=0      for each u in V[G]          if color[u]=white              DFS-VISIT(u)  DFS-VISIT(u)      color[u]=gray      d[u]=time=time+1      for each v in Adj[u]           if color[v]=white               father[v]=u               DFS-VISIT(v)      color[u]=black      f[u]=time=time+1 |

时间复杂度：

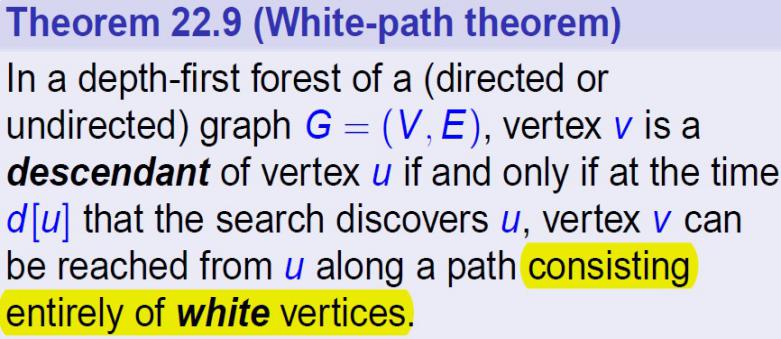


DFS的两个性质：

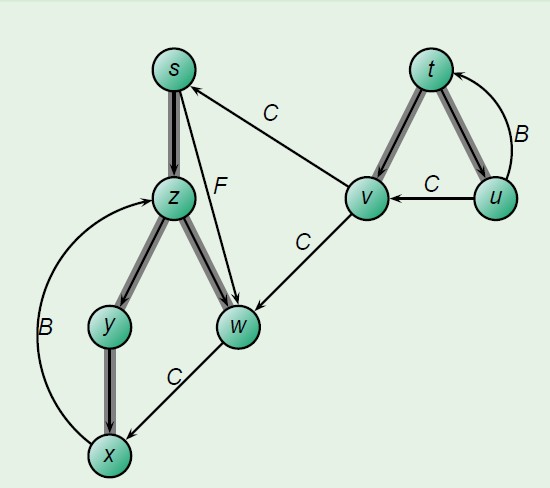
时间包含，子节点和完全包含是充要条件



白色路径定理，后面的证明会用到

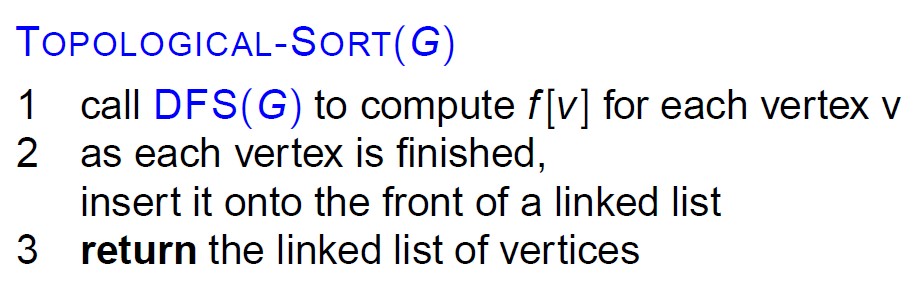


DFS中碰到的四种边的分类以及识别算法：

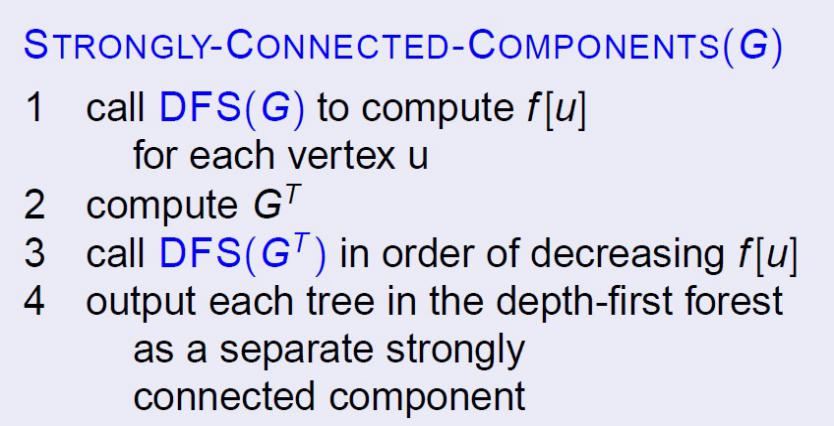


算法思想描述：可以对算法DFS做一些修改，使之遇到图中的边时，对其进行分类，算法核心思想在对于每条边u,v，当该边被第一次寻到时，根据所到达顶点的颜色，对其分类。  
1、白色的表明是树边  
2、灰色的表明是回边  
3、黑色的表明是正向边或者交叉边  
如果du<dv正向边,du>dv交叉边

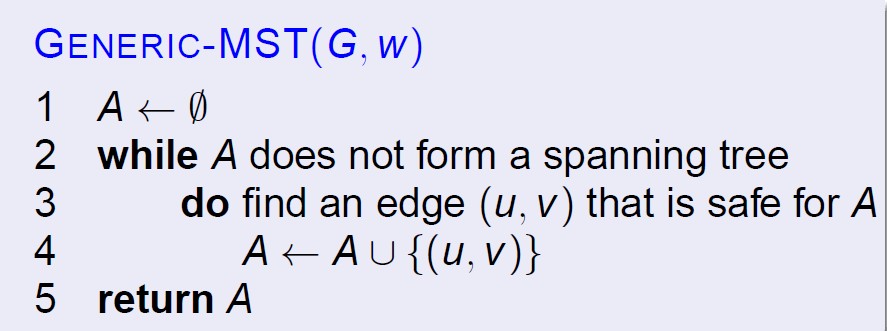
**6 拓扑排序**

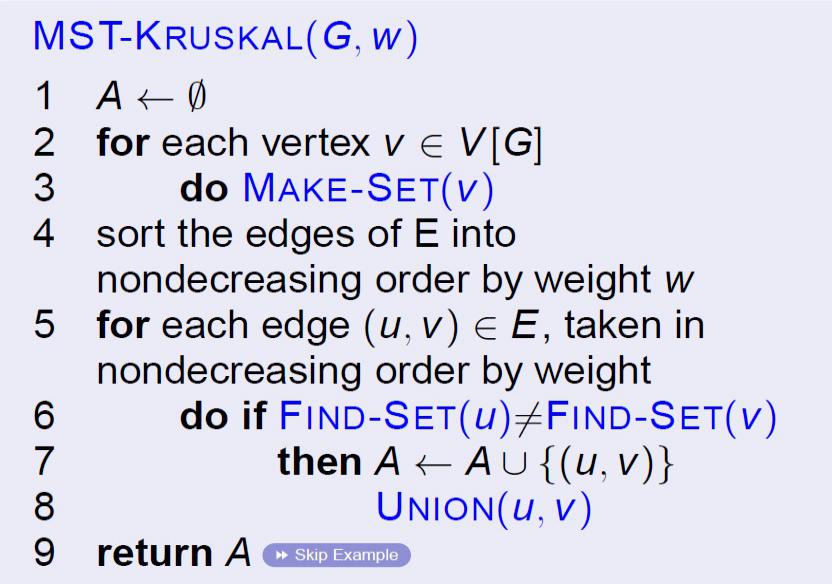


**7 强连通分量识别：**

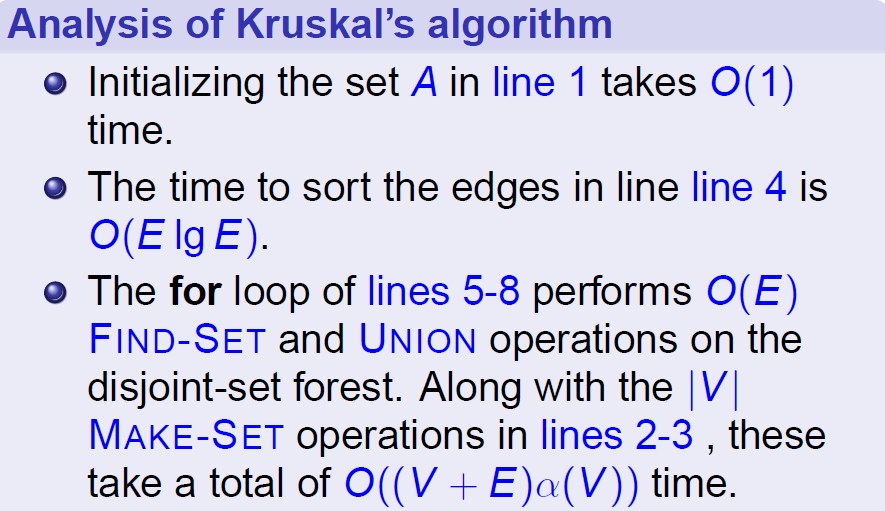


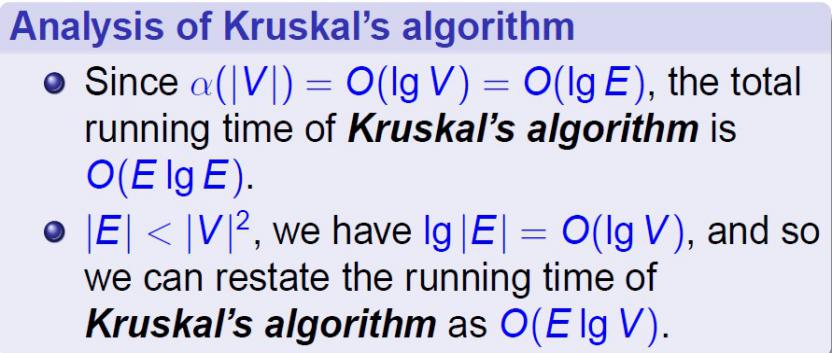
**8 最小生成树**

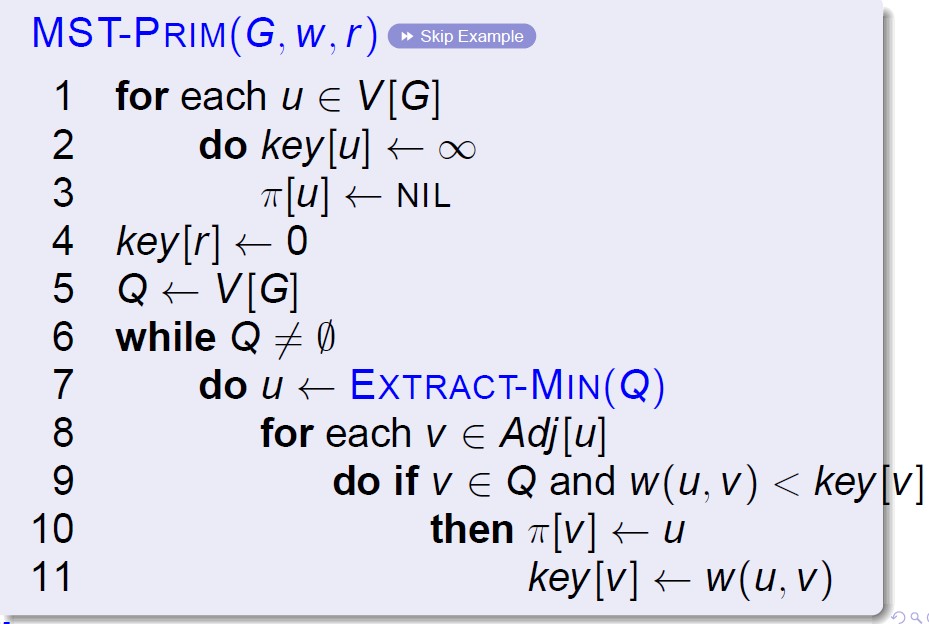




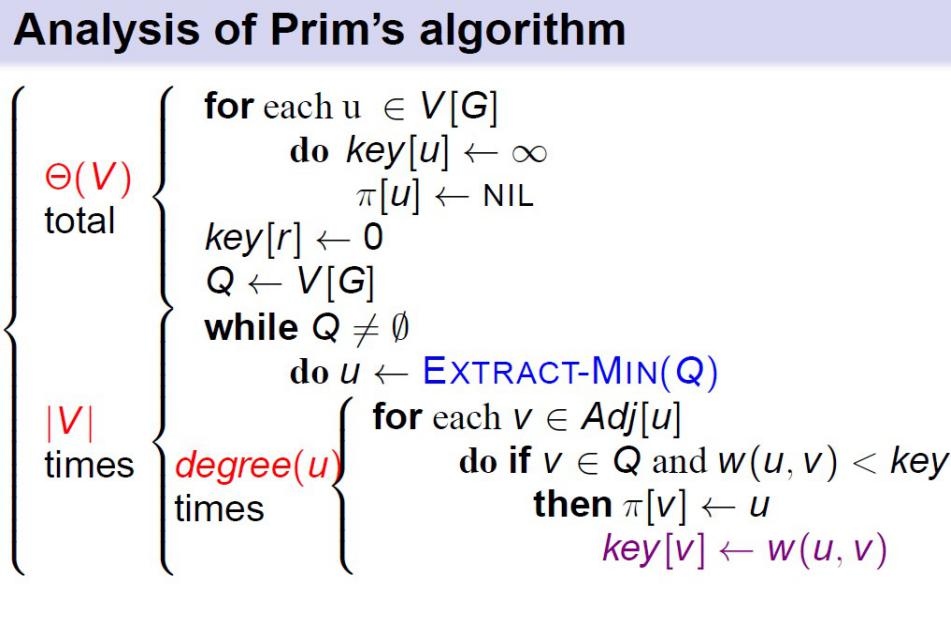
算法时间复杂度分析：

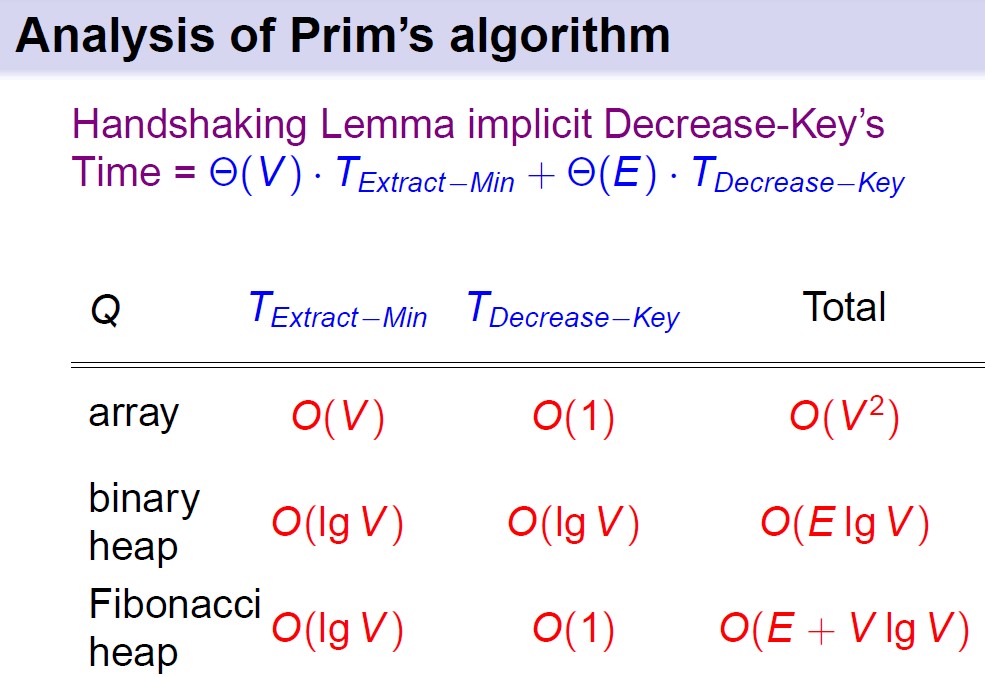






算法时间复杂度分析：





## 三、Gossip

**1 gossip概念**

顾名思义，类似于流言传播的概念，是一种可以按照自己的期望，自行选择与之交换信息的节点的通信方式

gossip, or anto-entropy, is an attractive way of replicating state that does not have strong consistency requirements

gossip 是一种弱一致[算法](http://lib.csdn.net/base/datastructure" \t "_blank" \o "算法与数据结构知识库)，也就是最终一致性算法。 Gossip[算法](http://lib.csdn.net/base/datastructure" \t "_blank" \o "算法与数据结构知识库)因为Cassandra而名声大噪，Gossip看似简单，但要真正弄清楚其本质远没看起来那么容易。  
特点： 在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致。每个节点可能知道所有其他节点，也可能仅知道几个邻居节点，只要这些节可以通过网络连通，最终他们的状态都是一致的，当然这也是疫情传播的特点。要注意到的一点是，即使有的节点因宕机而重启，有新节点加入，但经过一段时间后，这些节点的状态也会与其他节点达成一致，也就是说，Gossip天然具有分布式容错的优点。  
（1），去中心化，集群中各个节点都是对等的。   
（2），无法保证在某个时刻所有节点状态一致。   
（3），比较适合小数据量的同步。失败检测、路由同步、Pub/Sub、动态负载均衡

**2 Gossip是一个带冗余的容错算法**

更进一步，Gossip是一个最终一致性算法。虽然无法保证在某个时刻所有节点状态一致，但可以保证在”最终“所有节点一致，”最终“是一个现实中存在，但理论上无法证明的时间点。

因为Gossip不要求节点知道所有其他节点，因此又具有去中心化的特点，节点之间完全对等，不需要任何的中心节点。实际上Gossip可以用于众多能接受“最终一致性”的领域：失败检测、路由同步、Pub/Sub、动态负载均衡。但Gossip的缺点也很明显，冗余通信会对网路带宽、CUP资源造成很大的负载，而这些负载又受限于通信频率，该频率又影响着算法收敛的速度，后面我们会讲在各种场合下的优化方法。

**3 Gossip是一种去中心化、容错而又最终一致性的绝妙算法**，其收敛性不但得到证明还具有指数级的收敛速度。使用Gossip的系统可以很容易的把Server扩展到更多的节点，满足弹性扩展轻而易举。唯一的缺点是收敛是最终一致性，不使用那些强一致性的场景，比如2pc。

**4 算法描述**

假设有 {p, q, ...} 为协议参与者。 每个参与者都有关于一个自己信息的表。

用编程语言可以描述为：

记 InfoMap = Map<Key, (Value, Version)>， 那么每个参与者要维护一个 InfoMap 类型的变量 localInfo。 同时每一个参与者要知道所有其他参与者的信息, 即要维护一个全局的表，即 Map<participant, InfoMap> 类型的变量 globalMap。每个参与者更新自己的 localInfo， 而由 Gossip 协议负责将更新的信息同步到整个网络上。每个节点和系统中的某些节点成为 peer (如果系统的规模比较小，和系统中所有的其他节点成为 peer)。 有三种不同的同步信息的方法：

1）push-gossip: 最简单的情况下， 一个节点 p 向 q 发送整个 GlobalMap

2）pull-gossip: p 向 q 发送 digest, q 根据 digest 向 p 发送 p 过期的 (key, (value, version)) 列表

3）push-pull-gossip:与pull-gossip类似，只是多了一步，A再将本地比B新的数据推送给B，B更新本地

**5 Gossip节点的通信方式及收敛性**

**（1）根据原论文，两个节点（A、B）之间存在三种通信方式:**

**push:** A节点将数据(key,value,version)及对应的版本号推送给B节点，B节点更新A中比自己新的数据

**pull：**A仅将数据key,version推送给B，B将本地比A新的数据（Key,value,version）推送给A，A更新本地

**push/pull：**与pull类似，只是多了一步，A再将本地比B新的数据推送给B，B更新本地

如果把两个节点数据同步一次定义为一个周期，则在一个周期内，push需通信1次，pull需2次，push/pull则需3次，从效果上来讲，push/pull最好，理论上一个周期内可以使两个节点完全一致。直观上也感觉，push/pull的收敛速度是最快的。

假设每个节点通信周期都能选择（感染）一个新节点，则Gossip算法退化为一个二分查找过程，每个周期构成一个平衡二叉树，收敛速度为O(n2 )，对应的时间开销则为O(logn )。这也是Gossip理论上最优的收敛速度。但在实际情况中最优收敛速度是很难达到的，假设某个节点在第i个周期被感染的概率为pi ,第i+1个周期被感染的概率为pi+1

则pull的方式: pull

而push为：push

显然pull的收敛速度大于push，而每个节点在每个周期被感染的概率都是固定的p(0<p<1)，因此Gossip算法是基于p的平方收敛，也成为概率收敛，这在众多的一致性算法中是非常独特的。

**（2）Gossip的节点的工作方式又分两种：**

**Anti-Entropy（反熵）：**以固定的概率传播所有的数据

**Rumor-Mongering（谣言传播）：**仅传播新到达的数据

Anti-Entropy模式有完全的容错性，但有较大的网络、CPU负载；Rumor-Mongering模式有较小的网络、CPU负载，但必须为数据定义”最新“的边界，并且难以保证完全容错，对失败重启且超过”最新“期限的节点，无法保证最终一致性，或需要引入额外的机制处理不一致性。我们后续着重讨论Anti-Entropy模式的优化。

**6 Anti-Entropy的协调机制**

协调机制是讨论在每次2个节点通信时，如何交换数据能达到最快的一致性，也即消除两个节点的不一致性。上面所讲的push、pull等是通信方式，协调是在通信方式下的数据交换机制。协调所面临的最大问题是，因为受限于网络负载，不可能每次都把一个节点上的数据发送给另外一个节点，也即每个Gossip的消息大小都有上限。在有限的空间上有效率地交换所有的消息是协调要解决的主要问题。

在讨论之前先声明几个概念：

令N = {p,q,s,...}为需要gossip通信的server集合，有界大小

令(p1,p2,...)是宿主在节点p上的数据，其中数据有(key,value,version)构成，q的规则与p类似。为了保证一致性，规定数据的value及version只有宿主节点才能修改，其他节点只能间接通过Gossip协议来请求数据对应的宿主节点修改。

**（1）精确协调（Precise Reconciliation）**

精确协调希望在每次通信周期内都非常准确地消除双方的不一致性，具体表现为相互发送对方需要更新的数据，因为每个节点都在并发与多个节点通信，理论上精确协调很难做到。精确协调需要给每个数据项独立地维护自己的version，在每次交互是把所有的(key,value,version)发送到目标进行比对，从而找出双方不同之处从而更新。但因为Gossip消息存在大小限制，因此每次选择发送哪些数据就成了问题。当然可以随机选择一部分数据，也可确定性的选择数据。对确定性的选择而言，可以有最老优先（根据版本）和最新优先两种，最老优先会优先更新版本最新的数据，而最新更新正好相反，这样会造成老数据始终得不到机会更新，也即饥饿。

当然，开发这也可根据业务场景构造自己的选择算法，但始终都无法避免消息量过多的问题。

**（2）整体协调（Scuttlebutt Reconciliation）**

整体协调与精确协调不同之处是，整体协调不是为每个数据都维护单独的版本号，而是为每个节点上的宿主数据维护统一的version。比如节点P会为(p1,p2,...)维护一个一致的全局version，相当于把所有的宿主数据看作一个整体，当与其他节点进行比较时，只需必须这些宿主数据的最高version，如果最高version相同说明这部分数据全部一致，否则再进行精确协调。

整体协调对数据的选择也有两种方法：

广度优先：根据整体version大小排序，也称为公平选择

深度优先：根据包含数据多少的排序，也称为非公平选择。因为后者更有实用价值，所以原论文更鼓励后者

**7 使用场景**

**（1）[Redis](http://lib.csdn.net/base/redis" \t "_blank" \o "Redis知识库) 的 sentinel 的同步。 Cassandra集群。**

例子：有3个节点A，B，C。对任何一个节点A，以固定频率或一定的概率，将自己的数据及版本号发送到其他节点B，C。对于B，C，在接收到数据后会跟自己的数据进行对比，

将新数据保存下来，将A没有的数据发送给A，A可以在接收到数据后给B,C响应。

经过多次交互，最终达到一致状态

**（2）Cassandra集群**

经过验证，Cassandra实现了基于整体协调的push/push模式，有几个组件：

三条消息分别对应push/pull的三个阶段：

• GossipDigitsMessage

• GossipDigitsAckMessage

• GossipDigitsAck2Message

还有三种状态：

• EndpointState：维护宿主数据的全局version，并封装了HeartBeat和ApplicationState

• HeartBeat：心跳信息

• ApplicationState：系统负载信息（磁盘使用率）

Cassandra主要是使用Gossip完成三方面的功能：

• 失败检测

• 动态负载均衡

## 四、Longest Common Subsequence最长公共子序列

**1子序列的概念**

子序列不等于子串。子序列是一个字符串S去掉零个或者多个字符后所剩下的字符串就叫做子序列。最长公共子序列的意思就是寻找两个给定字符串的的子序列，该子序列在两个字符串中以相同的次序出现，但是不一定是连续的。（连续的那是子串）。

例如序列X=ABCBDAB，Y=BDCABA。序列BCA是X和Y的一个公共子序列，但是不是X和Y的最长公共子序列，子序列BCBA是X和Y的一个LCS，序列BDAB也是。最简单想到的寻找LCS的一种方法是枚举X所有的子序列，然后逐一检查是否是Y的子序列，并随时记录发现的最长子序列。假设X有m个元素，则X有2^m个子序列，指数级的时间，对长序列不实际。

**2 最长公共子序列是动态规划（DP）的典型例子。**

设X=<x1,x2,…,xm>和Y=<y1,y2,…,yn>为两个字符串，LCS(X,Y)表示X和Y的一个最长公共子序列，可以看出

如果xm=yn，则LCS ( X,Y ) = xm + LCS ( Xm-1,Yn-1 )。

如果xm!=yn，则LCS( X,Y )= max{ LCS ( Xm-1, Y ), LCS ( X, Yn-1 ) }

LCS问题也具有重叠子问题性质：为找出X和Y的一个LCS，可能需要找X和Yn-1的一个LCS以及Xm-1和Y的一个LCS。但这两个子问题都包含着找Xm-1和Yn-1的一个LCS，等等.

DP最终处理的还是数值（极值做最优解），找到了最优值，就找到了最优方案；为了找到最长的LCS，我们定义dp[i][j]记录序列LCS的长度，合法状态的初始值为当序列X的长度为0或Y的长度为0，公共子序列LCS长度为0，即dp[i][j]=0，所以用i和j分别表示序列X的长度和序列Y的长度，状态转移方程为

dp[i][j] = 0 如果i=0或j=0

dp[i][j] = dp[i-1][j-1] + 1 如果X[i-1] = Y[i-1] （X[i-1]就是X数组中的第i个）

dp[i][j] = max{ dp[i-1][j], dp[i][j-1] } 如果X[i-1] != Y[i-1]

#include <iostream>

using namespace std;

/\* LCS

\* 设序列长度都不超过20

\*/

int dp[21][21]; /\* 存储LCS长度, 下标i,j表示序列X,Y长度 \*/

char X[21];

char Y[21];

int i, j;

void main()

{

cin.getline(X,20);

cin.getline(Y,20);

int xlen = strlen(X);

int ylen = strlen(Y);

/\* dp[0-xlen][0] & dp[0][0-ylen] 都已初始化0 \*/

for(i = 1; i <= xlen; ++i)

{

for(j = 1; j <= ylen; ++j)

{

if(X[i-1] == Y[j-1])

{

dp[i][j] = dp[i-1][j-1] + 1;

}else if(dp[i][j-1] > dp[i-1][j])

{

dp[i][j] = dp[i][j-1];

}else

{

dp[i][j] = dp[i-1][j];

}

}

}

printf("len of LCS is: %d\n", dp[xlen][ylen]);

/\* 输出LCS 本来是逆序打印的，可以写一递归函数完成正序打印

这里采用的方法是将Y作为临时存储LCS的数组，最后输出Y

\*/

i = xlen;

j = ylen;

int k = dp[i][j];

char lcs[21] = {'\0'};

while(i && j)

{

if(X[i-1] == Y[j-1] && dp[i][j] == dp[i-1][j-1] + 1)

{

lcs[--k] = X[i-1];

--i; --j;

}else if(X[i-1] != Y[j-1] && dp[i-1][j] > dp[i][j-1])

{

--i;

}else

{

--j;

}

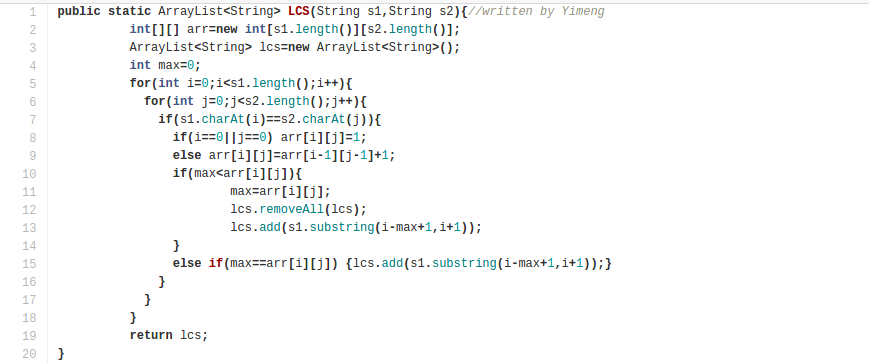
}

printf("%s\n",lcs);

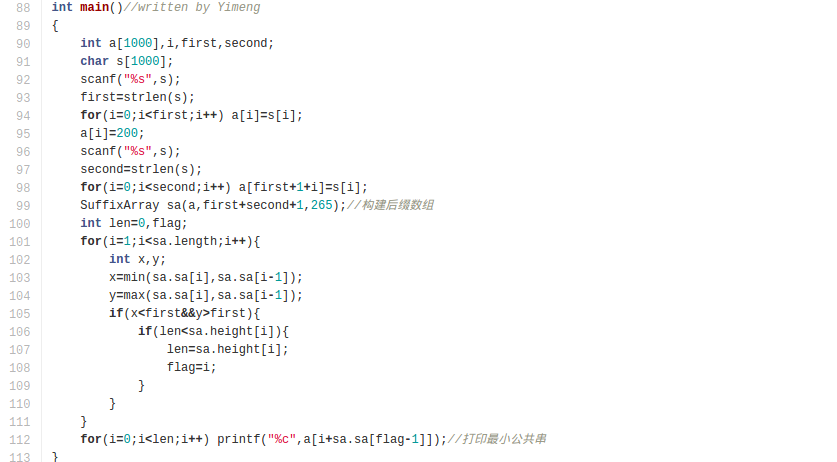
}

**3 字符串模式匹配基础上可以衍生出最长公共子串(Longest Common Substring)算法**

假设给出两个字符串S和N，找出二者之间的最大公共子串。按照正常思路可以很快想到：由大到小依次找出S的所有子串，每次将该子串作为Pattern使用Brute-Force算法搜索字符串N，如果找到匹配，便是最大公共子串。但是这种“算法”的时间复杂度达到O(n3)，严格意义上也不能称之为一种算法，充其量只能算一种人类思维，大部分情况下不应考虑选用这种算法解决问题。所谓“道不行，乘浮浮于海”。LCS值得推荐的算法有两种：使用动态编程(DP)或者使用后缀数组(suffix array)。先来看前者，这个算法应该算比较经典的。一般都是先以Fibonacci数列作为DP的入门讲解，之后就会借助这个算法进一步讲解DP的原理，所以这个算法很基本，需要熟练掌握。个人理解，DP本质就是递归，外加一个cache或者table而已。以DP的思维想问题，重点是状态转移方程。本算法的状态转移方程用代码可以表示为“if(i==0||j==0) arr[i][j]=1;else arr[i][j]=arr[i-1][j-1]+1;”理解后准确写出完整的算法代码不难，我写的Java代码如下：



如果这个算法很熟练了，可以进一步考虑最长公共子序列(Longest Common Subsequence)的算法，从而加深对这类DP问题的理解。DP方法的时间复杂度为平方阶O(n2)，空间复杂度也是O(n2)，效率有所提高，但却不是最高的算法。利用后缀数组的LCS算法与之相比效率更高，占用空间也更小。后缀数组是一个处理字符串非常好用的数据结构，本身结构不复杂，但是理解起来比较抽象。概念和理论不在这讨论，想深入了解的话建议阅读论文《后缀数组——处理字符串的有力工具》或其他相关资料。关于后最数组的应用，对sa[],rand[],height[]三个数组的公式关系要理解透彻。使用后缀数组处理字符串会使算法有质的飞跃。这种算法的本质是将求两个字符串的最长公共子串和两个字符串后缀的最长公共前缀的最大值等价，效率得到提升。后缀数组的求法比数组结构本身更重要，其方法有两种：倍增算法和DC3算法，时间复杂度分别为O(nlogn)和O(n)，前者简单一些，后者对我来说太难。我在网上找了一个后缀数组类的C++模版（完整代码），在使用该类的基础上写出的LCS算法如下：



**以上代码的时间复杂度为O(nlogn)，空间复杂度为O(n)，若使用DC3算法，其时间复杂度可以优化为线性阶O(n)，不得不由衷地感慨后缀数组之邪恶。**

**算法还是要以应用为主，毕竟不是学术大牛，不必过于执着于理论，不求甚解也许是更好的境界。最后结论，身为软件工程师，后缀数组算法至少要知其然，DP算法一定要知其所以然。**

## 五、Matrix Storage and Operations矩阵存储运算

**1 矩阵存储运算原理**

矩阵就是线性代数中的那个矩阵，在计算机中常用二维数组来存储它，然后与一些操作函数封装在一起构成矩阵类。

**2 矩阵存储运例子**

矩阵也可以用一维数组来存储，但是要注意矩阵下标与一维数组下标间的转换关系。将这样的数组与矩阵的存，删，改，查等操作一封装就构成了矩阵类。

**@对称矩阵**

对称矩阵就是关于矩阵对角线对称的矩阵，由于具有对称的特性，因此在存储时利用可减少存储空间，设n阶矩阵的行列为i,j，在矩阵存储表示时，根据i>j ? index=(i-1)\*n+(j-1)：index=(j-1)\*n+(i-1)求出在一维数组中的下标，然后存进去就行了。取的时候也一样。

**@稀疏矩阵**

稀疏矩阵就是0比较多，因此可直接存储非0项，可定义一个结构体包含三个属性，分别是非零项在矩阵中的row,col,value，然后用一个顺序表的结构来把这些结构体串起来，添上一些标记矩阵信息的属性和一些矩阵操作，就构成了稀疏矩阵类。但是顺序表在查找，删除时比较麻烦。而用有行指针的链表来表示就很方便了，它包含了一个行指针数组，所有该行的矩阵非零元都被用指针attach到记录该行的数组元素上。当然你也可以设计一个含行指针和列指针数组的链表稀疏矩阵。

**3 问题**

　矩阵在计算机图形学中有着重要的地位，刚开始学习时很容易被矩阵乘法的先后顺序所困扰，因为这里存在很多因素，比如什么是列矩阵和行矩阵，两种矩阵在做变换时会有什么差别？什么是列优先和行优先？矢量左乘矩阵和右乘矩阵是怎么定义的？

**4 列矩阵和行矩阵**

　列矩阵（column major）和行矩阵（row major）是数学上的概念，和电脑无关，它只是一套约定（convention），按照矢量和矩阵的乘法运算时，矢量是列矢还是行矢命名，这里只说4x4矩阵。齐次矢量可以看成是一个1x4的矩阵，就是行矢；或者4x1的矩阵，就是列矢。列矩阵变换矢量时，用矩阵乘以列矢量（M \* v），如下。其中1、2、3为三个轴向矢量。

　| x1 x2 x3 xt | | x |

　| y1 y2 y3 yt | \* | y |

　| z1 y2 y3 zt | | z |

　| 0 0 0 1 | | w |

列矩阵之间乘法组合时，组合顺序为从右到左。例如依次做放缩S、旋转R、平移T，则组合为T \* R \* S。行矩阵变换矢量时，用行矢量乘以矩阵（v \* M），如下。其中1、2、3为三个轴向矢量。

　| x1 y1 z1 0 |

　| x y z w | \* | x2 y2 z2 0 |

　 | x3 y3 z3 0 |

　| xt yt zt 1 |

　行矩阵之间乘法组合时，组合顺序为从左到右。例如依次做放缩S、旋转R、平移T，则组合为S \* R \* T。

5 **列矩阵和行矩阵**

　 列优先存储（column major storage）和行优先存储（row major storage）是计算机上的概念，指多维数组的元素在内存中的排列顺序。

　 对应到矩阵上，按内存地址顺序编号，则列优先存储的顺序为

　　| 1 5 9 13 |

　　| 2 6 10 14 |

　　| 3 7 11 15 |

　　| 4 8 12 16 |

　　行优先存储的顺序为

　　| 1 2 3 4 |

　　| 5 6 7 8 |

　　| 9 10 11 12 |

　　| 13 14 15 16 |

**6 左乘和右乘**

左乘(left multiplication, pre-multication)和右乘(right multiplication, post-multication)，这里的左右、pre/post，指的是矢量和矩阵相乘时，矢量的位置。对标量和矩阵相乘时，指标量的位置。说简单点,左乘(又称前乘)就是乘在左边（即乘号前）,右乘（又称后乘）就是乘在右边（即乘号后）.比如说,A左乘E即AE

**7 Opengl和D3D中的矩阵**

Opengl采用的是列优先，D3D采用的是行优先。 抛开不同底层实现的差异，我们在进行模型视图变换时，一般的顺序都是先缩放，然后旋转，最后进行平移，这样才能达到正确的变换效果。因此对于opengl，V`=T\*R\*S\*V,但是对于D3D，V`=V\*S\*R\*T。

**8 .OSG中的矩阵**

osg的底层为opengl,但是在其矩阵实现时，是按照行矩阵来实现，矩阵的内部存储使用一个二维数组，这样在glLoadMatrix访问的刚好是osg的矩阵的的转置,可以这样理解这个区别：对于opengl，V`=M\*V，根据矩阵运算法则：V`转置 = V转置 \* M转置，矢量的转置不影响我们的使用，这样我们可以在osg中把矩阵当成行矩阵对待，因此在变换顺序上为S \* R \* T，这样也更符合从左到右的习惯。

## 六、Minimum Spanning Tree最小生成树

**1 最小生成树**

给定一个无向图，如果他的某个子图中，任意两个顶点都能互相连通并且是一棵树，那么这棵树就叫做生成树（spanning tree）.如果边上有权值，那么使得边权和最小的生成树叫做最小生成树（MST，Minimum Spanning Tree）

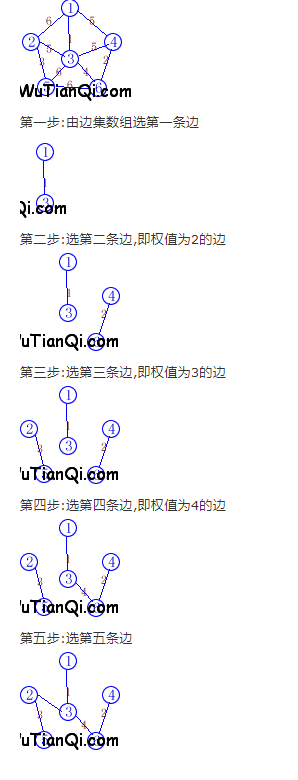
**（1）克鲁斯卡尔算法**

图的存贮结构采用边集数组,且权值相等的边在数组中排列次序可以是任意的.该方法对于边相对比较多的不是很实用,浪费时间.

**（2）普里姆算法**

图的存贮结构采用邻接矩阵.此方法是按各个顶点连通的步骤进行,需要用一个顶点集合,开始为空集,以后将以连通的顶点陆续加入到集合中,全部顶点加入集合后就得到所需的最小生成树 .

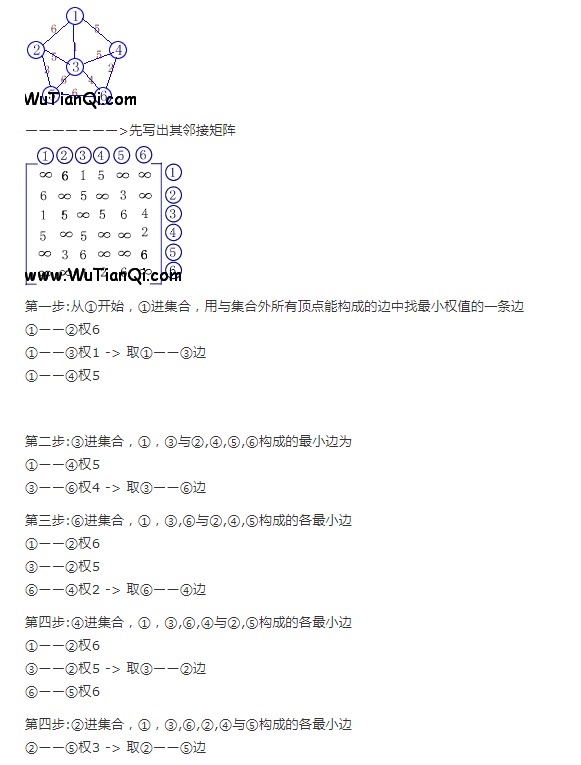
**2 克鲁斯卡尔算法**

方法:将图中边按其权值由小到大的次序顺序选取,若选边后不形成回路,则保留作为一条边,若形成回路则除去.依次选够(n-1)条边,即得最小生成树.(n为顶点数)  
  


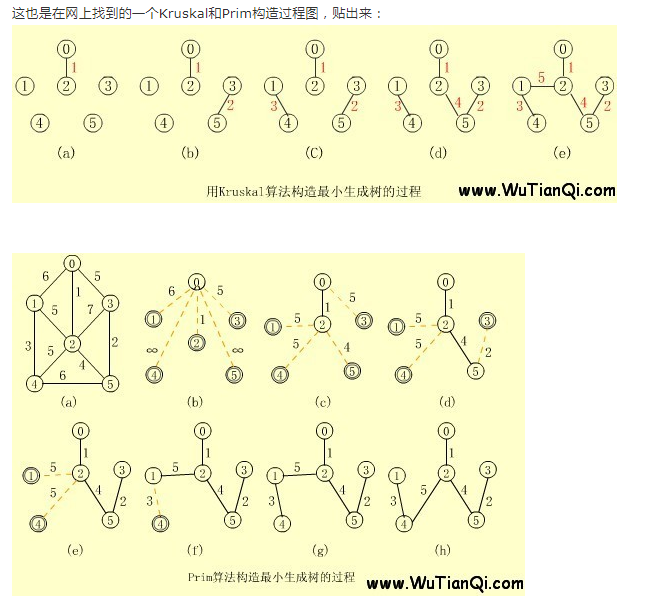
**3 普里姆算法**

方法:从指定顶点开始将它加入集合中,然后将集合内的顶点与集合外的顶点所构成的所有边中选取权值最小的一条边作为生成树的边,并将集合外的那个顶点加入到集合中,表示该顶点已连通.再用集合内的顶点与集合外的顶点构成的边中找最小的边,并将相应的顶点加入集合中,如此下去直到全部顶点都加入到集合中,即得最小生成树.

例在下图中从1点出发求出此图的最小生成树,并按生成树的边的顺序将顶点与权值填入表中.



**4 两种方法示例**



**5 使用场景**

**（1）城市公交网建设问题**

【问题描述】

有一张城市地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权为在这两个城市之间修建高速公路的造价，研究后发现，这个地图有一个特点，即任一对城市都是连通的。现在的问题是，要修建若干高速公路把所有城市联系起来，问如何设计可使得工程的总造价最少？

【输入格式】

n（城市数，1<=n<=100）

　　e（边数）

　　以下e行，每行3个数i,j,wij，表示在城市i,j之间修建高速公路的造价。

【输出格式】

　　n-1行，每行为两个城市的序号，表明这两个城市间建一条高速公路。

【输入样例】

　　5 8

　　1 2 2

　　2 5 9

　　5 4 7

　　4 1 10

　　1 3 12

　　4 3 6

　　5 3 3

　　2 3 8

【输出样例】

　　　1 2

　　　2 3

　　　3 4

　　　3 5

## Paxos

**1数据库高可用性难题**

数据库的数据一致和持续可用对电子商务和互联网金融的意义不言而喻，而这些业务在使用数据库时，无论 MySQL 还是 Oracle，都会面临一个艰难的取舍，就是如何处理主备库之间的数据同步。对于传统的主备模式或者一主多备模式，我们都需要考虑的问题，就是与备机保持强同步还是异步复制。

对于强同步模式，要求主机必须把 Redolog 同步到备机之后，才能应答客户端，一旦主备之间出现网络抖动，或者备机宕机，则主机无法继续提供服务，这种模式实现了数据的强一致，但是牺牲了服务的可用性，且由于跨机房同步延迟过大使得跨机房的主备模式也变得不实用。而对于异步复制模式，主机写本地成功后，就可以立即应答客户端，无需等待备机应答，这样一旦主机宕机无法启动，少量不同步的日志将丢失，这种模式实现了服务持续可用，但是牺牲了数据一致性。这两种方式对应的就是 Oracle 的 Max Protection 和 Max Performance 模式，而 oracle另一个最常用的 Max Availability 模式，则是一个折中，在备机无应答时退化为 Max Performance 模式，我认为本质上还是异步复制。主备模式还有一个无法绕过的问题，就是选主，最简单山寨的办法，搞一个单点，定时 Select 一下主机和各个备机，貌似 MHA 就是这个原理，具体实现细节我就不太清楚了。一个改进的方案是使用类似 ZooKeeper 的多点服务替代单点，各个数据库机器上使用一个 Agent 与单点保持 Lease，主机 Lease 过期后，立即置为只读。改进的方案基本可以保证不会出现双主，而缺点是 ZooKeeper 的可维护性问题，以及多级 Lease 的恢复时长问题

**2 Paxos 协议简单回顾**

主备方式处理数据库高可用问题有上述诸多缺陷，要改进这种数据同步方式，我们先来梳理下数据库高可用的几个基本需求：

数据不丢失

服务持续可用

自动的主备切换

使用Paxos协议的日志同步可以实现这三个需求，而 Paxos 协议需要依赖一个基本假设，主备之间有多数派机器（N / 2 + 1）存活并且他们之间的网络通信正常，如果不满足这个条件，则无法启动服务，数据也无法写入和读取。

我们先来简单回顾一下 Paxos 协议的内容，首先，Paxos 协议是一个解决分布式系统中，多个节点之间就某个值（提案）达成一致（决议）的通信协议。它能够处理在少数派离线的情况下，剩余的多数派节点仍然能够达成一致。然后，再来看一下协议内容，它是一个两阶段的通信协议，推导过程我就不写了（中文资料请参考这篇 Http://t.cn/R40lGrp ），直接看最终协议内容：

**（1）第一阶段 Prepare**

P1a：Proposer 发送 Prepare

Proposer 生成全局唯一且递增的提案 ID（Proposalid，以高位时间戳 + 低位机器 IP 可以保证唯一性和递增性），向 Paxos 集群的所有机器发送 PrepareRequest，这里无需携带提案内容，只携带 Proposalid 即可。

P1b：Acceptor 应答 Prepare

Acceptor 收到 PrepareRequest 后，做出“两个承诺，一个应答”。

两个承诺：

第一，不再应答 Proposalid 小于等于（注意：这里是 <= ）当前请求的 PrepareRequest；

第二，不再应答 Proposalid 小于（注意：这里是 < ）当前请求的 AcceptRequest

一个应答：

返回自己已经 Accept 过的提案中 ProposalID 最大的那个提案的内容，如果没有则返回空值;

注意：这“两个承诺”中，蕴含两个要点：

就是应答当前请求前，也要按照“两个承诺”检查是否会违背之前处理 PrepareRequest 时做出的承诺；

应答前要在本地持久化当前 Propsalid。

**（2）第二阶段 Accept**

P2a：Proposer 发送 Accept

“提案生成规则”：Proposer 收集到多数派应答的 PrepareResponse 后，从中选择proposalid最大的提案内容，作为要发起 Accept 的提案，如果这个提案为空值，则可以自己随意决定提案内容。然后携带上当前 Proposalid，向 Paxos 集群的所有机器发送 AccpetRequest。

P2b：Acceptor 应答 Accept

Accpetor 收到 AccpetRequest 后，检查不违背自己之前作出的“两个承诺”情况下，持久化当前 Proposalid 和提案内容。最后 Proposer 收集到多数派应答的 AcceptResponse 后，形成决议。

**3 Basic Paxos 同步日志的理论模型**

上面是 Lamport 提出的算法理论，那么 Paxos 协议如何具体应用在 Redolog 同步上呢，我们先来看最简单的理论模型，就是在 N 个 Server的机群上，持久化数据库或者文件系统的操作日志，并且为每条日志分配连续递增的 LogID，允许多个客户端并发的向机群内的任意机器发送日志同步请求。在这个场景下，不同 Logid 标识的日志都是一个个相互独立的 Paxos Instance，每条日志独立执行完整的 Paxos 两阶段协议。

因此在执行 Paxos 之前，需要先确定当前日志的 Logid，理论上对每条日志都可以从 1 开始尝试，直到成功持久化当前日志，但是为了降低失败概率，可以先向集群内的 Acceptor 查询他们 PrepareResponse 过的最大 Logid，从多数派的应答结果中选择最大的 Logi-d，加 1 后，作为本条日志的 Logid。然后以当前 Logid 标识 Paxos Instance，开始执行Paxos两阶段协议。可能出现的情况是，并发情况下，当前 Logid 被其他日志使用，那么在 P2a 阶段确定的提案内容可能就不是自己本次要同步的日志内容，这种情况下，就要重新决定logid，然后重新开始执行 Paxos 协议。

考虑几种异常情况，Proposer 在 P1b 或 P2b 阶段没有收到多数派应答，可能是受到了其他 Logid 相同而 Proposalid 更大的 Proposer 干扰，或者是网络、机器等问题，这种情况下则要使用相同的 Logid，和新生成的 Proposalid 来重新执行 Paxos 协议。恢复时，按照 Logid 递增的顺序，针对每条日志执行完整 Paxos 协议成功后，形成决议的日志才可以进行回放。那么问题来了：比如 A/B/C 三个 Server，一条日志在 A/B 上持久化成功，已经形成多数派，然后B宕机；另一种情况，A/B/C 三个 Server，一条日志只在A 上持久化成功，超时未形成多数派，然后B宕机。上述两种情况，最终的状态都是 A 上有这条日志，C 上没有，那么应该怎么处理呢？

这里提一个名词：“最大 Commit 原则”，这个阳振坤博士给我讲授 Paxos 时提出的名词，我觉得它是 Paxos 协议的最重要隐含规则之一，一条超时未形成多数派应答的提案，我们即不能认为它已形成决议，也不能认为它未形成决议，跟“薛定谔的猫”差不多，这条日志是“又死又活”的，只有当你观察它（执行 Paxos 协议）的时候，你才能得到确定的结果。因此对于上面的问题，答案就是无论如何都对这条日志重新执行 Paxos。这也是为什么在恢复的时候，我们要对每条日志都执行 Paxos 的原因。

**4 Multi Paxos 的实际应用**

上述 Basic-Paxos 只是理论模型，在实际工程场景下，比如数据库同步 Redolog，还是需要集群内有一个 leader，作为数据库主机，和多个备机联合组成一个 Paoxs 集群，对 Redolog 进行持久化。此外持久化和回放时每条日志都执行完整 Paxos 协议（3 次网络交互，2 次本地持久化），代价过大，需要优化处理。因此使用 Multi-Paxos 协议，要实现如下几个重要功能：

自动选主

简化同步逻辑

简化回放逻辑

我在刚刚学习 Paxos 的时候，曾经认为选主就是跑一轮 Paxos 来形成“谁是 leader”的决议，其实并没有这么简单，因为 Paxos 协议的基本保证就是一旦形成决议，就不能更改，那么再次选新主就没办法处理了。因此对“选主”，需要变通一下思路，还是执行 Paxos 协议，但是我们并不关心决议内容，而是关心“谁成功得到了多数派的 AcceptResponse”，这个 Server 就是选主产生的 Leader。而多轮选主，就是针对同一个 Paxos Instance 反复执行，最后赢得多数派 Accept 的 Server 就是“当选 Leader”。

不幸的是执行 Paxos 胜出的“当选 Leader”还不能算是真正的 Leader，只能算是“当选 Leader”，就像美国总统一样，“当选总统”是赢得选举的总统，但是任期还未开始他还不是真正的总统。在 Multi-Paxos 中因为可能存在多个 Server 先后赢得了选主，因此新的“当选leader”还要立即写出一条日志，以确认自己的 Leader 身份。这里就顺势引出日志同步逻辑的简化，我们将 Leader 选主看作 Paxos 的 Prepare 阶段，这个 Prepare 操作在逻辑上一次性的将后续所有即将产生的日志都执行 Prepare，因此在 Leader任期内的日志同步，都使用同一个 Proposalid，只执行 Accept 阶段即可。那么问题来了，各个备机在执行 Accept 的时候，需要注意什么？

答案是上面提到过的“两个承诺”，因为我们已经把选主的那轮 Paxos 看做 Prepare 操作了，所以对于后续要 Accept 的日志，要遵守“两个承诺”。所以，对于先后胜出选主的多个“当选 Leader”，他们同步日志时携带的 Proposalid 的大小是不同的，只有最大的 Pro-posalid 能够同步日志成功，成为正式的 Leader。

再进一步简化，选主 Leader 后，“当选 Leader”既然必先写一条日志来确认自己的 Leader身份，而协议允许多个“当选 Leader”产生，那么选主过程的本质其实就是为了拿到各个备机的“两个承诺”而已，选主过程本身产生的决议内容并没有实际意义，所以可以进一步简化为只执行 Prepare 阶段，而无需执行 Accept。

再进一步优化，与 Raft 协议不同，Multi-Paxos 并不要求新任 Leader 本地拥有全部日志，因此新任 Leader 本地可能与其他 Server 相差了一些日志，它需要知道自己要补全哪些日志，因此它要向多数派查询各个机器上的 MaxLogD，以确定补全日志的结束 LogID。这个操作成为 GetMaxLogID，我们可以将这个操作与选主的 Prepare 操作搭车一起发出。这个优化并非 Multi-Paxos 的一部分，只是一个工程上比较有效的实现。

回放逻辑的简化就比较好理解了，Leader 对每条形成多数派的日志，异步的写出一条“确认日志”即可，回放时如果一条日志拥有对应的“确认日志”，则不需要重新执行 Paoxs，直接回放即可。对于没有“确认日志”的，则需要重新执行 Paxos。工程上为了避免“确认日志”与对应的 Redolog 距离过大而带来回放的复杂度，往往使用滑动窗口机制来控制他们的距离。同时“确认日志”也用来提示备机可以回放收到的日志了。与 Raft 协议不同，由于 Multi-Paxos 允许日志不连续的确认（请思考：不连续确认的优势是什么？），以及允许任何成员都可以当选 Leader，因此新任 leader 需要补全自己本地缺失的日志，以及对未“确认”的日志重新执行 Paxos。我把这个过程叫做日志的“重确认”，本质上就是按照“最大commit原则”，使用当前最新的 Proposalid，逐条的对这些日志重新执行 Paxos，成功后再补上对应的“确认日志”。

相对于 Raft 连续确认的特性，使用 Multi-Paxos 同步日志，由于多条日志间允许乱序确认，理论上会出现一种被称我们团队同学戏称为“幽灵复现”的诡异现象，如下图所示（图片引用自我的博客）

第一轮中A被选为 Leader，写下了 1-10 号日志，其中 1-5 号日志形成了多数派，并且已给客户端应答，而对于 6-10 号日志，客户端超时未能得到应答。

第二轮，A 宕机，B 被选为 Leader，由于 B 和 C 的最大的 LogID 都是 5，因此 B 不会去重确认 6 - 10 号日志，而是从 6 开始写新的日志，此时如果客户端来查询的话，是查询不到上一轮 6 - 10 号 日志内容的，此后第二轮又写入了 6 - 20 号日志，但是只有 6 号和 20 号日志在多数派。

第三轮，A 又被选为 Leader，从多数派中可以得到最大 LogID 为 20，因此要将 7 - 20 号日志执行重确认，其中就包括了 A 上的 7-10 号日志，之后客户端再来查询的话，会发现上次查询不到的 7 - 10 号日志又像幽灵一样重新出现了。

处理“幽灵复现”问题，需要依赖新任 Leader 在完成日志重确认，开始写入新的 Redolog 之前，写出一条被称为 StartWorking 的日志，这条日志的内容中记录了当前 Leader 的 EpochID（ 可以使用 Proposalid 的值），并且 Leader 每写一条日志都在日志内容中携带现任 Leader 的 EpochID。回放时，经过了一条 StartWorking 日志之后，再遇到 EpochID 比它小的日志，就直接忽略掉，比如按照上面例子画出的这张图，7 - 19 号日志要在回放时被忽略掉。

**5 依赖时钟误差的变种 Paxos 选主协议简单分析**

阿里的阳振坤老师根据 Paxos 协议设计了一个简化版本的选主协议，相对 MultiPaxos 和 Raft 协议的优势在于，它不需要持久化任何数据，引入选主窗口的概念，使得大部分场景下集群内的所有机器能够几乎同时发起选主请求，便于投票时比对预定的优先级。下面的图引用自 OB 团队在公开场合分享 PPT 中的图片。

如图所示，选主协议规定选主窗口开启是当前时间对一个T取余为0的时间，即只能在第 0，T，2T，3T...N\*T 的时间点上开启选主窗口，协议将一次选主划分为三个阶段

T1 预投票开始即由各个选举组成员向集群里的其他机器发送拉票请求；

一段时间后进入 T2 预投票开始，选举组各个成员根据接受到的拉票请，从中选出优先级最高的，给它投票应答；一段时间后进入 T3 计票阶段，收到多数派投票的成员成为 leader，并向投票组其他成员发送自己上任的消息。

假设时钟误差最大为 Tdiff，网络网路传输单程最长耗时为 Tst

收到预投票消息的时间区间 [T1 - Tdiff × 2，T1 + Tdiff × 2 + Tst = T2]

收到投票消息的时间区间 [T2 - Tdiff×2，T2 + Tdiff × 2 + Tst = T3]

收到广播消息的时间区间 [T3 - Tdiff×2，T3 + Tdiff × 2 + Tst = T4]

选主耗时 Telect = T4-T1 = Tdiff × 6 + Tst × 3

因此最差情况下，选主开始后，经过 Tdiff × 6 + Tst × 3 的 d 时间，就可以选出 Leader 各个成员投出选票后，就从自己的 T1 时刻开始计时，认为 leader 持续 lease 时间内有效，在 Lease 有效期内，Leader 每隔 Telect 的时间就向其他成员发出续约请求，将 Lease 时间顺延一个 Telect，如果 Lease 过期后 Leader 没有续约，则各个成员等待下一个选主窗口到来后发起选主。因此最差情况下的无主时间是：Lease 时间 + Telect + 选主窗口间隔时间 T。

这个选主算法相对 Paxos 和 Raft 更加简单，但是对时钟误差有比较强的依赖，时钟误差过大的情况下，会造成投票分裂无法选出主，甚至可能出现双主（不过话说任何保持 Leader 身份的 Lease 机制都得依赖时钟…），因此可能仅仅适合 BAT 这种配备了原子钟和 GPS 校准时钟，能够控制时钟误差在 100ms 以内的土豪机房。2015 年闰秒时，这个选主算法已经上线至支付宝，当时测试了几个月吧，1 秒的跳变已经太大，当时测试了几个月，修改 ntp 配置缓慢校准，最后平稳渡过。

**6 使用场景**

**（1）、ZooKeeper 所使用的 zad 协议与 Paxos 协议有什么区别？**

Zab 用的是Epoch 和 Count 的组合来唯一表示一个值, 而Raft 用的是 Term和 Index.

Zab 的 Follower 在投票给一个 Leader 之前必须和 Leader 的日志达成一致,而 Raft的 Follower 则简单地说是谁的 Term 高就投票给谁。

Raft 协 议的心跳是从 Leader 到 Follower, 而 zab 协议则相反。

Raft 协议数据只有单向地从 Leader 到 Follower (成为 Leader 的条件之一就是拥有最新的 Log), 而 Zab 协议在 Discovery 阶段, 一个 Prospective Leader 需要将自己的Log 更新为 Quorum 里面最新的 Log,然后才好在 Synchronization 阶段将 Quorum 里的其他机器的 Log 都同步到一致。

**（2）、Paxos 能完成在全球同步的业务吗？理论上支持多少机器同步?**

Paxos 成员组横跨全球的案例我还没有见过 Paper，我个人认为它并不适合全球不同，原因是延迟太大，但是 Google 的 Spanner 和 Amazon 的 Aurora 都实现了横跨北美多 IDC 的同步；理论上多少都行，你能接受延迟就可以。

**（3）、问个问题，能否简单说说 Raft 算法和 Paxos 算法的异同？应用场的异同？**

Raft 可以认为是一种简化的 Multi-Paxos 实现，他的最大简化之处在于备机接受 Leader 日志的前提是收到 LogID 连续的日志，在这个假设前提下，没有我文中提到的“幽灵复现”和“重确认”问题。简化带来的代价是对网络抖动的容忍度稍低一些，考虑这样的场景 ABC 三台机器，C 临时下线一会错过一些日志，然后 C上 线了，但是在 C 补全日志之前，AB 如果再宕机一台的话，服务就停了。

**（4）、Paxos 实现是独立的库或服务还是和具体的业务逻辑绑定，上线前如何验证 Paxos 算法实现的正确性？**

OB 实现的 Paxos 是和事务 Redolog 库比较紧耦合的，没有独立的库；测试方案一个是 Monkey tests，随机模拟各种异常环境，包括断网、网络延迟、机器宕机、包重复到达等情况保持压力和异常；另外一个是做了一个简易的虚拟机，来解释测试 Case，通过人工构造多种极端的场景，来是系统立即进入一个“梦境”。

（**5）、Logid 和 proposalid都应该是不能重复的，这个是如何保证的？原子钟的精确性仅仅是为了选主吗？**

首先，Leader 任期内，Logid 只由 Leader 产生，没有重复性的问题；

第二，Leader 产生后，会执行 GetMaxLogID，从集群多数派拿到最大的 Logid，加以后作为本届任期内的 Logid 起点，这也可以保证有效日志 logid 不重复。Proposalid，高位使用 64 位时间戳，低位使用 IP 地址，可以保证唯一性和递增性。

**（6）、在用 Paxos 协议做 Master 和 Slave 一致性保证时，Paxos 日志回放应该怎样去做？**

Master 形成多数派确认后，异步的写出“确认日志”，Slave 回放到确认日志之后，才能去回放收到的正常日志。因此一般情况下，备机总是要落后主机一点点的。

## 八、Raft

**1 Raft是由Stanford提出的一种更易理解的一致性算法**

1. **意在取代目前广为使用的Paxos算法。**

目前，在各种主流语言中都有了一些开源实现，比如本文中将使用的基于JGroups的Raft协议实现。关于Raft的原理，强烈推荐动画版Raft讲解。

**（2）在Raft中，每个结点会处于下面三种状态中的一种：**

**follower：**所有结点都以follower的状态开始。如果没收到leader消息则会变成candidate状态

**candidate：**会向其他结点“拉选票”，如果得到大部分的票则成为leader。这个过程就叫做Leader选举(Leader Election)

**leader：**所有对系统的修改都会先经过leader。每个修改都会写一条日志(log entry)。leader收到修改请求后的过程如下，这个过程叫做日志复制(Log Replication)：

复制日志到所有follower结点(replicate entry)

大部分结点响应时才提交日志

通知所有follower结点日志已提交

所有follower也提交日志

**2 Leader Election**

当follower在选举超时时间(election timeout)内未收到leader的心跳消息(append entries)，则变成candidate状态。为了避免选举冲突，这个超时时间是一个150~300ms之间的随机数。成为candidate的结点发起新的选举期(election term)去“拉选票”：

重置自己的计时器

投自己一票

发送 Request Vote消息

如果接收结点在新term内没有投过票那它就会投给此candidate，并重置它自己的选举超时时间。candidate拉到大部分选票就会成为leader，并定时发送心跳——Append Entries消息，去重置各个follower的计时器。当前Term会继续直到某个follower接收不到心跳并成为candidate。

如果不巧两个结点同时成为candidate都去“拉票”怎么办？这时会发生Splite Vote情况。两个结点可能都拉到了同样多的选票，难分胜负，选举失败，本term没有leader。之后又有计时器超时的follower会变成candidate，将term加一并开始新一轮的投票。

**3 Log Replication**

当发生改变时，leader会复制日志给follower结点，这也是通过Append Entries心跳消息完成的。前面已经列举了Log Replication的过程，这里就不重复了。

Raft能够正确地处理网络分区（“脑裂”）问题。假设A~E五个结点，B是leader。如果发生“脑裂”，A、B成为一个子分区，C、D、E成为一个子分区。此时C、D、E会发生选举，选出C作为新term的leader。这样我们在两个子分区内就有了不同term的两个leader。这时如果有客户端写A时，因为B无法复制日志到大部分follower所以日志处于uncommitted未提交状态。而同时另一个客户端对C的写操作却能够正确完成，因为C是新的leader，它只知道D和E。

当网络通信恢复，B能够发送心跳给C、D、E了，却发现“改朝换代”了，因为C的term值更大，所以B自动降格为follower。然后A和B都回滚未提交的日志，并从新leader那里复制最新的日志。但这样是不是就会丢失更新？

**4 JGroups中的Raft**

JGroups是Java里比较流行的网络通信框架，近期顺应潮流，它也推出了Raft基于JGroups的实现。简单试用了一下，还比较容易上手，底层Raft的内部机制都被API屏蔽掉了。下面就通过一个分布式计数器的实例来学习一下Raft协议在JGroups中的实际用法。

Maven依赖如下：

<dependency>

<groupId>org.jgroups</groupId>

<artifactId>jgroups-raft</artifactId>

<version>0.2</version>

</dependency>

其实JGroups-raft的Jar包中已经自带了一个Counter的Demo，但仔细看了一下，有的地方写的有些麻烦，不太容易把握住Raft这根主线。所以这里就参照官方的例子，进行了简写，突出Raft协议的基本使用方法。JGroups-raft目前资料不多，InfoQ上的这篇文章很不错，还有官方文档。

**5核心API**

使用JGroups-raft时，我们一般会实现两个接口：RAFT.RoleChange和StateMachine：

实现RAFT.RoleChange接口的方法能通知我们当前哪个结点是leader

实现StateMachine执行要实现一致性的操作

典型单点服务实现方式就是：

JChannel ch = null;

RaftHandle handle = new RaftHandle(ch, this);

handle.addRoleListener(role -> {

if(role == Role.Leader)

// start singleton services

else

// stop singleton services

});

**6 默认配置**

jgroups-raft.jar中已经带了一个raft.xml配置文件，作为实例程序我们可以直接使用它。

简要解释一下最核心的几个配置项，参照GitHub上的文档：

UDP：IP多播配置

raft.NO\_DUPES：是否检测新加入结点的ID与老结点有重复

raft.ELECTION：选举超时时间的随机化范围

raft.RAFT：所有Raft集群的成员必须在这里声明，也可以在运行时通过addServer/removeServer动态修改

raft.REDIRECT：是否转发请求给leader

raft.CLIENT：在哪个IP和端口上接收客户端请求

<config xmlns="urn:org:jgroups"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/jgroups.xsd">

<UDP

mcast\_addr="228.5.5.5"

mcast\_port="${jgroups.udp.mcast\_port:45588}"

... />

<raft.NO\_DUPES/>

<raft.ELECTION election\_min\_interval="100" election\_max\_interval="500"/>

<raft.RAFT members="A,B,C" raft\_id="${raft\_id:undefined}"/>

<raft.REDIRECT/>

<raft.CLIENT bind\_addr="0.0.0.0" />

</config>

**7 JGroupsRaftTest**

JGroupsRaftTest的职责主要有三个：

创建Raft协议的JChannel

创建CounterService

循环读取用户输入

目前简单实现了几种操作包括：初始化计数器、加一、减一、读取计数器、查看Raft日志、做Raft快照（用于压缩日志文件）等。其中对计数器的操作，因为要与其他Raft成员进行分布式通信，所以当前集群必须要多于一个结点时才能进行操作。如果要支持单结点时的操作，需要做特殊处理。

import org.jgroups.JChannel;

import org.jgroups.protocols.raft.RAFT;

import org.jgroups.util.Util;

/\*\*

\* Test jgroups raft algorithm implementation.

\*/

public class JGroupsRaftTest {

private static final String CLUSTER\_NAME = "ctr-cluster";

private static final String COUNTER\_NAME = "counter";

private static final String RAFT\_XML = "raft.xml";

public static void main(String[] args) throws Exception {

JChannel ch = new JChannel(RAFT\_XML).name(args[0]);

CounterService counter = new CounterService(ch);

try {

doConnect(ch, CLUSTER\_NAME);

doLoop(ch, counter);

} finally {

Util.close(ch);

}

}

private static void doConnect(JChannel ch, String clusterName) throws Exception {

ch.connect(clusterName);

}

private static void doLoop(JChannel ch, CounterService counter) {

boolean looping = true;

while (looping) {

int key = Util.keyPress("\n[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit\n" +

"first-applied=" + ((RAFT) ch.getProtocolStack().findProtocol(RAFT.class)).log().firstApplied() +

", last-applied=" + counter.lastApplied() +

", commit-index=" + counter.commitIndex() +

", log size=" + Util.printBytes(counter.logSize()) + ": ");

if ((key == '0' || key == '1' || key == '2') && !counter.isLeaderExist()) {

System.out.println("Cannot perform cause there is no leader by now");

continue;

}

long val;

switch (key) {

case '0':

counter.getOrCreateCounter(COUNTER\_NAME, 1L);

break;

case '1':

val = counter.incrementAndGet(COUNTER\_NAME);

System.out.printf("%s: %s\n", COUNTER\_NAME, val);

break;

case '2':

val = counter.decrementAndGet(COUNTER\_NAME);

System.out.printf("%s: %s\n", COUNTER\_NAME, val);

break;

case '3':

counter.dumpLog();

break;

case '4':

counter.snapshot();

break;

case 'x':

looping = false;

break;

case '\n':

System.out.println(COUNTER\_NAME + ": " + counter.get(COUNTER\_NAME) + "\n");

break;

}

}

}

}

**8 CounterService**

CounterService是我们的核心类，利用Raft实现了分布式的计数器操作，它的API主要由四部分组成：

Raft Local API：操作本地Raft的状态，像日志大小、做快照等

Raft API：实现Raft的监听器和状态机的方法

roleChanged：本地Raft的角色发生变化

apply：分布式通信消息

readContentFrom/writeContentTo：读写快照

Counter API：计数器的分布式API

Counter Native API：计数器的本地API。直接使用的话相当于脏读

import org.jgroups.Channel;

import org.jgroups.protocols.raft.RAFT;

import org.jgroups.protocols.raft.Role;

import org.jgroups.protocols.raft.StateMachine;

import org.jgroups.raft.RaftHandle;

import org.jgroups.util.AsciiString;

import org.jgroups.util.Bits;

import org.jgroups.util.ByteArrayDataInputStream;

import org.jgroups.util.ByteArrayDataOutputStream;

import org.jgroups.util.Util;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

class CounterService implements StateMachine, RAFT.RoleChange {

private RaftHandle raft;

private final Map<String, Long> counters;

private enum Command {

CREATE, INCREMENT\_AND\_GET, DECREMENT\_AND\_GET, GET, SET

}

public CounterService(Channel ch) {

this.raft = new RaftHandle(ch, this);

this.counters = new HashMap<>();

raft.raftId(ch.getName())

.addRoleListener(this);

}

public int lastApplied() {

return raft.lastApplied();

}

public int commitIndex() {

return raft.commitIndex();

}

public int logSize() {

return raft.logSize();

}

public void dumpLog() {

System.out.println("\nindex (term): command\n---------------------");

raft.logEntries((entry, index) -> {

StringBuilder log = new StringBuilder()

.append(index)

.append(" (").append(entry.term()).append("): ");

if (entry.command() == null ) {

System.out.println(log.append("<marker record>"));

return;

} else if (entry.internal()) {

System.out.println(log.append("<internal command>"));

return;

}

ByteArrayDataInputStream in = new ByteArrayDataInputStream(

entry.command(), entry.offset(), entry.length()

);

try {

Command cmd = Command.values()[in.readByte()];

String name = Bits.readAsciiString(in).toString();

switch (cmd) {

case CREATE:

log.append(cmd)

.append("(").append(name).append(", ")

.append(Bits.readLong(in))

.append(")");

break;

case GET:

case INCREMENT\_AND\_GET:

case DECREMENT\_AND\_GET:

log.append(cmd)

.append("(").append(name).append(")");

break;

default:

throw new IllegalArgumentException("Command " + cmd + "is unknown");

}

System.out.println(log);

}

catch (IOException e) {

throw new IllegalStateException("Error when dump log", e);

}

});

System.out.println();

}

public void snapshot() {

try {

raft.snapshot();

} catch (Exception e) {

throw new IllegalStateException("Error when snapshot", e);

}

}

public boolean isLeaderExist() {

return raft.leader() != null;

}

@Override

public void roleChanged(Role role) {

System.out.println("roleChanged to: " + role);

}

@Override

public byte[] apply(byte[] data, int offset, int length) throws Exception {

ByteArrayDataInputStream in = new ByteArrayDataInputStream(data, offset, length);

Command cmd = Command.values()[in.readByte()];

String name = Bits.readAsciiString(in).toString();

System.out.println("[" + new SimpleDateFormat("HH:mm:ss.SSS").format(new Date())

+ "] Apply: cmd=[" + cmd + "]");

long v1, retVal;

switch (cmd) {

case CREATE:

v1 = Bits.readLong(in);

retVal = create0(name, v1);

return Util.objectToByteBuffer(retVal);

case GET:

retVal = get0(name);

return Util.objectToByteBuffer(retVal);

case INCREMENT\_AND\_GET:

retVal = add0(name, 1L);

return Util.objectToByteBuffer(retVal);

case DECREMENT\_AND\_GET:

retVal = add0(name, -1L);

return Util.objectToByteBuffer(retVal);

default:

throw new IllegalArgumentException("Command " + cmd + "is unknown");

}

}

@Override

public void readContentFrom(DataInput in) throws Exception {

int size = in.readInt();

System.out.println("ReadContentFrom: size=[" + size + "]");

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

AsciiString name = Bits.readAsciiString(in);

Long value = Bits.readLong(in);

counters.put(name.toString(), value);

}

}

@Override

public void writeContentTo(DataOutput out) throws Exception {

synchronized (counters) {

int size = counters.size();

System.out.println("WriteContentFrom: size=[" + size + "]");

out.writeInt(size);

for (Map.Entry<String, Long> entry : counters.entrySet()) {

AsciiString name = new AsciiString(entry.getKey());

Long value = entry.getValue();

Bits.writeAsciiString(name, out);

Bits.writeLong(value, out);

}

}

}

public void getOrCreateCounter(String name, long initVal) {

Object retVal = invoke(Command.CREATE, name, false, initVal);

counters.put(name, (Long) retVal);

}

public long incrementAndGet(String name) {

return (long) invoke(Command.INCREMENT\_AND\_GET, name, false);

}

public long decrementAndGet(String name) {

return (long) invoke(Command.DECREMENT\_AND\_GET, name, false);

}

public long get(String name) {

return (long) invoke(Command.GET, name, false);

}

private Object invoke(Command cmd, String name, boolean ignoreRetVal, long... values) {

ByteArrayDataOutputStream out = new ByteArrayDataOutputStream(256);

try {

out.writeByte(cmd.ordinal());

Bits.writeAsciiString(new AsciiString(name), out);

for (long val : values) {

Bits.writeLong(val, out);

}

byte[] rsp = raft.set(out.buffer(), 0, out.position());

return ignoreRetVal ? null : Util.objectFromByteBuffer(rsp);

}

catch (IOException ex) {

throw new RuntimeException("Serialization failure (cmd="

+ cmd + ", name=" + name + ")", ex);

}

catch (Exception ex) {

throw new RuntimeException("Raft set failure (cmd="

+ cmd + ", name=" + name + ")", ex);

}

}

public synchronized Long create0(String name, long initVal) {

counters.putIfAbsent(name, initVal);

return counters.get(name);

}

public synchronized Long get0(String name) {

return counters.getOrDefault(name, 0L);

}

public synchronized Long add0(String name, long delta) {

Long oldVal = counters.getOrDefault(name, 0L);

return counters.put(name, oldVal + delta);

}

}

**9 运行测试**

我们分别以A、B、C为参数，启动三个JGroupsRaftTest服务。这样会自动在C:\Users\cdai\AppData\Local\Temp下生成A.log、B.log、C.log三个日志文件夹。

cdai@vm /cygdrive/c/Users/cdai/AppData/Local/Temp

$ tree A.log/ B.log/ C.log/

A.log/

|-- 000005.sst

|-- 000006.log

|-- CURRENT

|-- LOCK

|-- LOG

|-- LOG.old

`-- MANIFEST-000004

B.log/

|-- 000003.log

|-- CURRENT

|-- LOCK

|-- LOG

`-- MANIFEST-000002

C.log/

|-- 000003.log

|-- CURRENT

|-- LOCK

|-- LOG

`-- MANIFEST-000002

0 directories, 17 files

**10 分布式一致性**

首先A创建计数器，B“加一”，C“减一”。可以看到尽管我们是分别在A、B、C上执行这三个操作，但三个结点都先后（leader提交日志后通知follower）通过apply()方法收到消息，并在本地的计数器Map上同步执行操作，保证了数据的一致性。最后停掉A服务，可以看到B通过roleChanged()得到消息，提升为新的Leader，并与C一同继续提供服务。

A的控制台输出：

-------------------------------------------------------------------

GMS: address=A, cluster=ctr-cluster, physical address=2001:0:9d38:6abd:cbb:1f78:3f57:50f6:50100

-------------------------------------------------------------------

[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit

first-applied=0, last-applied=0, commit-index=0, log size=0b:

roleChanged to: Candidate

roleChanged to: Leader

0

[14:16:00.744] Apply: cmd=[CREATE]

[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit

first-applied=0, last-applied=1, commit-index=1, log size=1b:

[14:16:07.002] Apply: cmd=[INCREMENT\_AND\_GET]

[14:16:14.264] Apply: cmd=[DECREMENT\_AND\_GET]

3

index (term): command

---------------------

1 (29): CREATE(counter, 1)

2 (29): INCREMENT\_AND\_GET(counter)

3 (29): DECREMENT\_AND\_GET(counter)

B的控制台输出：

-------------------------------------------------------------------

GMS: address=B, cluster=ctr-cluster, physical address=2001:0:9d38:6abd:cbb:1f78:3f57:50f6:50101

-------------------------------------------------------------------

[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit

first-applied=0, last-applied=0, commit-index=0, log size=0b:

[14:16:01.300] Apply: cmd=[CREATE]

1

counter: 2

[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit

first-applied=0, last-applied=2, commit-index=1, log size=2b:

[14:16:07.299] Apply: cmd=[INCREMENT\_AND\_GET]

[14:16:14.304] Apply: cmd=[DECREMENT\_AND\_GET]

roleChanged to: Candidate

roleChanged to: Leader

C的控制台输出：

-------------------------------------------------------------------

GMS: address=C, cluster=ctr-cluster, physical address=2001:0:9d38:6abd:cbb:1f78:3f57:50f6:55800

-------------------------------------------------------------------

[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit

first-applied=0, last-applied=0, commit-index=0, log size=0b:

[14:16:01.300] Apply: cmd=[CREATE]

[14:16:07.299] Apply: cmd=[INCREMENT\_AND\_GET]

2

counter: 3

[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit

first-applied=0, last-applied=3, commit-index=2, log size=3b:

[14:16:14.304] Apply: cmd=[DECREMENT\_AND\_GET]

**11 服务恢复**

在只有B和C的集群中，我们执行了一次“加一”。当我们重新启动A服务时，它会自动执行这条日志，保持与B和C的一致。从日志的index能够看出，69是一个Term，也就是A为Leader时的“任期”，而70也就是B为Leader时。

A的控制台输出：

-------------------------------------------------------------------

GMS: address=A, cluster=ctr-cluster, physical address=2001:0:9d38:6abd:cbb:1f78:3f57:50f6:53237

-------------------------------------------------------------------

[0] Create [1] Increment [2] Decrement [3] Dump log [4] Snapshot [x] Exit

first-applied=0, last-applied=3, commit-index=3, log size=3b:

[14:18:45.275] Apply: cmd=[INCREMENT\_AND\_GET]

[14:18:45.277] Apply: cmd=[GET]

3

index (term): command

---------------------

1 (69): CREATE(counter, 1)

2 (69): INCREMENT\_AND\_GET(counter)

3 (69): DECREMENT\_AND\_GET(counter)

4 (70): INCREMENT\_AND\_GET(counter)

5 (70): GET(counter)

**12 Paxos 算法与 Raft 算法的差异**

Raft强调是唯一leader的协议，此leader至高无上

Raft：新选举出来的leader拥有全部提交的日志，而 Paxos 需要额外的流程从其他节点获取已经被提交的日志，它允许日志有空洞

相同点：得到大多数的赞成，这个 entries 就会定下来，最终所有节点都会赞成

NWR模型

N： N个备份

W：要写入至少 w 份才认为成功

R : 至少读取 R 个备份

W+ R > N ——> R > N - W(未更新成功的) ，代表每次读取，都至少读取到一个最新的版本（更新成功的），从而不会读到一份旧数据

问题：并非强一致性，会出现一些节点上的数据并不是最新版本，但却进行了最新的操作

版本冲突问题：矢量钟 Vector Clock ： 谁更新的我，我的版本号是什么（对于同一个操作者的同一操作，版本号递增）

## 九、Sort Arithmetic

**1排序概念**

有内部排序和外部排序，内部排序是数据记录在内存中进行排序，而外部排序是因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存。我们这里说说八大排序就是内部排序。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、堆排序或归并排序序。

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；



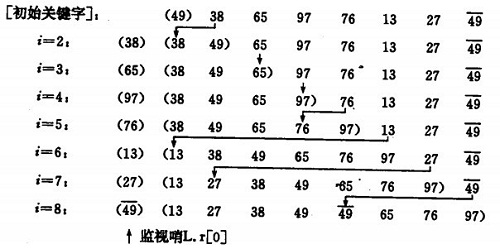
**2 插入排序—直接插入排序(Straight Insertion Sort)**

基本思想:

将一个记录插入到已排序好的有序表中，从而得到一个新，记录数增1的有序表。即：先将序列的第1个记录看成是一个有序的子序列，然后从第2个记录逐个进行插入，直至整个序列有序为止。

要点：设立哨兵，作为临时存储和判断数组边界之用。

直接插入排序示例：



如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳定的。

算法的实现：

package com;

public class InsertSort {

public static void main(String[] args) {

int a[] = {3,1,5,7,2,4,9,6,10,8};

InsertSort obj=new InsertSort();

System.out.println("初始值：");

obj.print(a);

obj.insertSort(a);

System.out.println("\n排序后：");

obj.print(a);

}

public void print(int a[]){

for(int i=0;i<a.length;i++){

System.out.print(a[i]+" ");

}

}

public void insertSort(int[] a) {

for(int i=1;i<a.length;i++){//从头部第一个当做已经排好序的，把后面的一个一个的插到已经排好的列表中去。

if(a[i]<a[i-1]){

int j;

int x=a[i];//x为待插入元素

a[i]=a[i-1];

for(j=i-1; j>=0 && x<a[j];j--){//通过循环，逐个后移一位找到要插入的位置。

a[j+1]=a[j];

}

a[j+1]=x;//插入

}

}

}

}

效率：

时间复杂度：O（n^2）.

其他的插入排序有二分插入排序，2-路插入排序。

**3 插入排序—希尔排序（Shell`s Sort）**

@希尔排序是1959 年由D.L.Shell 提出来的，相对直接排序有较大的改进。希尔排序又叫缩小增量排序

@基本思想：

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

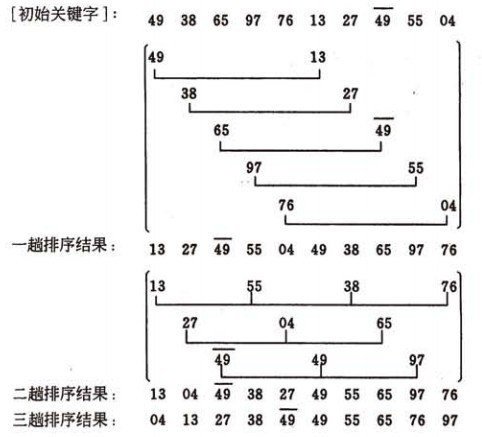
@操作方法：

选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

按增量序列个数k，对序列进行k 趟排序；

每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

@希尔排序的示例：



算法实现：

我们简单处理增量序列：增量序列d = {n/2 ,n/4, n/8 .....1} n为要排序数的个数

即：先将要排序的一组记录按某个增量d（n/2,n为要排序数的个数）分成若干组子序列，每组中记录的下标相差d.对每组中全部元素进行直接插入排序，然后再用一个较小的增量（d/2）对它进行分组，在每组中再进行直接插入排序。继续不断缩小增量直至为1，最后使用直接插入排序完成排序。

package com;

/\*

\* Java实现希尔排序（缩小增量排序）

\* author:wyr

\* 2016-7-14

\*两个步骤：1，建堆 2，对顶与堆的最后一个元素交换位置

\*/

public class ShellSort {

public static void main(String[] args) {

int a[] = {3,1,5,7,2,4,9,6,10,8};

ShellSort obj=new ShellSort();

System.out.println("初始值：");

obj.print(a);

obj.shellSort(a);

System.out.println("\n排序后：");

obj.print(a);

}

private void shellSort(int[] a) {

int dk = a.length/2;

while( dk >= 1 ){

ShellInsertSort(a, dk);

dk = dk/2;

}

}

private void ShellInsertSort(int[] a, int dk) {//类似插入排序，只是插入排序增量是1，这里增量是dk,把1换成dk就可以了

for(int i=dk;i<a.length;i++){

if(a[i]<a[i-dk]){

int j;

int x=a[i];//x为待插入元素

a[i]=a[i-dk];

for(j=i-dk; j>=0 && x<a[j];j=j-dk){//通过循环，逐个后移一位找到要插入的位置。

a[j+dk]=a[j];

}

a[j+dk]=x;//插入

}

}

}

public void print(int a[]){

for(int i=0;i<a.length;i++){

System.out.print(a[i]+" ");

}

}

}

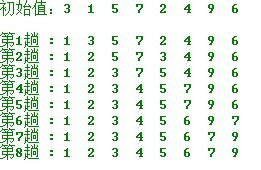
希尔排序时效分析很难，关键码的比较次数与记录移动次数依赖于增量因子序列d的选取，特定情况下可以准确估算出关键码的比较次数和记录的移动次数。目前还没有人给出选取最好的增量因子序列的方法。增量因子序列可以有各种取法，有取奇数的，也有取质数的，但需要注意：增量因子中除1 外没有公因子，且最后一个增量因子必须为1。希尔排序方法是一个不稳定的排序方法。

**4 选择排序—简单选择排序（Simple Selection Sort）**

@基本思想：

在要排序的一组数中，选出最小（或者最大）的一个数与第1个位置的数交换；然后在剩下的数当中再找最小（或者最大）的与第2个位置的数交换，依次类推，直到第n-1个元素（倒数第二个数）和第n个元素（最后一个数）比较为止。

@简单选择排序的示例：



@操作方法：

第一趟，从n 个记录中找出关键码最小的记录与第一个记录交换；

第二趟，从第二个记录开始的n-1 个记录中再选出关键码最小的记录与第二个记录交换；

以此类推.....

第i 趟，则从第i 个记录开始的n-i+1 个记录中选出关键码最小的记录与第i 个记录交换，

直到整个序列按关键码有序。

@算法实现：

package com;

/\*

\* Java实现希尔排序（缩小增量排序）

\* author:wyr

\* 2016-7-14

\*两个步骤：1，建堆 2，对顶与堆的最后一个元素交换位置

\*/

public class SimpleSelectSort {

public static void main(String[] args) {

int a[] = {3,1,5,7,2,4,9,6,10,8};

SimpleSelectSort obj=new SimpleSelectSort();

System.out.println("初始值：");

obj.print(a);

obj.selectSort(a);

System.out.println("\n排序后：");

obj.print(a);

}

private void selectSort(int[] a) {

for(int i=0;i<a.length;i++){

int k=i;//k存放最小值下标。每次循环最小值下标+1

for(int j=i+1;j<a.length;j++){//找到最小值下标

if(a[k]>a[j])

k=j;

}

swap(a,k,i);//把最小值放到它该放的位置上

}

}

public void print(int a[]){

for(int i=0;i<a.length;i++){

System.out.print(a[i]+" ");

}

}

public void swap(int[] data, int i, int j) {

if (i == j) {

return;

}

data[i] = data[i] + data[j];

data[j] = data[i] - data[j];

data[i] = data[i] - data[j];

}

}

**5 简单选择排序的改进——二元选择排序**

简单选择排序，每趟循环只能确定一个元素排序后的定位。我们可以考虑改进为每趟循环确定两个元素（当前趟最大和最小记录）的位置,从而减少排序所需的循环次数。改进后对n个数据进行排序，最多只需进行[n/2]趟循环即可。具体实现如下：

void SelectSort(int r[],int n) {

int i ,j , min ,max, tmp;

for (i=1 ;i <= n/2;i++) {

// 做不超过n/2趟选择排序

min = i; max = i ; //分别记录最大和最小关键字记录位置

for (j= i+1; j<= n-i; j++) {

if (r[j] > r[max]) {

max = j ; continue ;

}

if (r[j]< r[min]) {

min = j ;

}

}

//该交换操作还可分情况讨论以提高效率

tmp = r[i-1]; r[i-1] = r[min]; r[min] = tmp;

tmp = r[n-i]; r[n-i] = r[max]; r[max] = tmp;

}

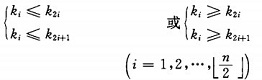
}

**6 选择排序—堆排序（Heap Sort）**

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

基本思想：

堆的定义如下：具有n个元素的序列（k1,k2,...,kn),当且仅当满足

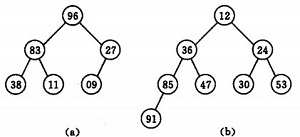


时称之为堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最小项（小顶堆）。

若以一维数组存储一个堆，则堆对应一棵完全二叉树，且所有非叶结点的值均不大于(或不小于)其子女的值，根结点（堆顶元素）的值是最小(或最大)的。如：

（a）大顶堆序列：（96, 83,27,38,11,09)

(b) 小顶堆序列：（12，36，24，85，47，30，53，91）



初始时把要排序的n个数的序列看作是一棵顺序存储的二叉树（一维数组存储二叉树），调整它们的存储序，使之成为一个堆，将堆顶元素输出，得到n 个元素中最小(或最大)的元素，这时堆的根节点的数最小（或者最大）。然后对前面(n-1)个元素重新调整使之成为堆，输出堆顶元素，得到n 个元素中次小(或次大)的元素。依此类推，直到只有两个节点的堆，并对它们作交换，最后得到有n个节点的有序序列。称这个过程为堆排序。

因此，实现堆排序需解决两个问题：

1. 如何将n 个待排序的数建成堆；

2. 输出堆顶元素后，怎样调整剩余n-1 个元素，使其成为一个新堆。

首先讨论第二个问题：输出堆顶元素后，对剩余n-1元素重新建成堆的调整过程。

调整小顶堆的方法：

1）设有m 个元素的堆，输出堆顶元素后，剩下m-1 个元素。将堆底元素送入堆顶（（最后一个元素与堆顶进行交换），堆被破坏，其原因仅是根结点不满足堆的性质。

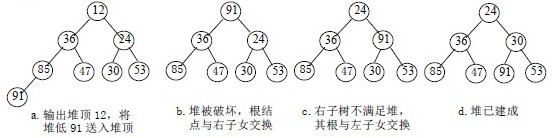
2）将根结点与左、右子树中较小元素的进行交换。

3）若与左子树交换：如果左子树堆被破坏，即左子树的根结点不满足堆的性质，则重复方法 （2）.

4）若与右子树交换，如果右子树堆被破坏，即右子树的根结点不满足堆的性质。则重复方法 （2）.

5）继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建成。

称这个自根结点到叶子结点的调整过程为筛选。如图：



再讨论对n 个元素初始建堆的过程。

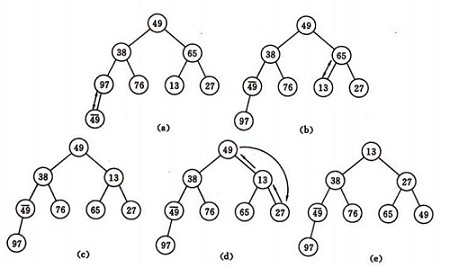
建堆方法：对初始序列建堆的过程，就是一个反复进行筛选的过程。

1）n 个结点的完全二叉树，则最后一个结点是第个结点的子树。

2）筛选从第个结点为根的子树开始，该子树成为堆。

3）之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。

如图建堆初始过程：无序序列：（49，38，65，97，76，13，27，49）



算法的实现：

从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

package com;

/\*

\* Java实现快速排序算法

\* 由大到小排序

\* author:wyr

\* 2016-7-14

\*两个步骤：1，建堆 2，对顶与堆的最后一个元素交换位置

\*/

public class HeapSort {

public static void main(String[] args) {

int a[] = {3,1,5,7,2,4,9,6,10,8};

HeapSort obj=new HeapSort();

System.out.println("初始值：");

obj.print(a);

for(int i=0;i<a.length;i++){

obj.createLittleHeap(a,a.length-1-i);//创建堆,创建的是小顶堆。每次循环完，二叉树的根节点都是最小值，所以与此时的未排好部分最后一个值交换位置

obj.swap(a, 0, a.length - 1 - i);//与最后一个值交换位置，最小值找到了位置

obj.print(a);

System.out.println();

}

System.out.println("\n排序后：");

obj.print(a);

}

/\*

\* 创建小顶堆：双亲节点小于子节点的值。从叶子节点开始，直到根节点。这样建立的堆定位最小值

\*/

private void createLittleHeap(int[] data, int last) {

for (int i = (last- 1) / 2; i >= 0; i--) { //找到最后一个叶子节点的双亲节点

// 保存当前正在判断的节点

int parent = i;

// 若当前节点的左子节点存在，即子节点存在

while (2 \* parent + 1 <= last) {

// biggerIndex总是记录较大节点的值,先赋值为当前判断节点的左子节点

int bigger = 2 \* parent + 1;//bigger指向左子节点

if (bigger < last) { //说明存在右子节点

if (data[bigger] > data[bigger+ 1]) { //右子节点>左子节点时

bigger=bigger+1;

}

}

if (data[parent] > data[bigger]) { //若双亲节点值大于子节点中最大的

// 若当前节点值比子节点最大值小，则交换2者得值，交换后将biggerIndex值赋值给k

swap(data, parent, bigger);

parent = bigger;

} else {

break;

}

}

}

}

public void print(int a[]){

for(int i=0;i<a.length;i++){

System.out.print(a[i]+" ");

}

}

public void swap(int[] data, int i, int j) {

if (i == j) {

return;

}

data[i] = data[i] + data[j];

data[j] = data[i] - data[j];

data[i] = data[i] - data[j];

}

}

堆排序需要双层循环，第一层控制循环多少次，第二层得到每次的最小值（小顶堆）

package arrayTest;

import java.util.ArrayList;

public class Solution32 {

/\* 输入n个整数，找出其中最小的K个数。

\* 例如输入4,5,1,6,2,7,3,8这8个数字，

\* 则最小的4个数字是1,2,3,4,。

\* \*/

public ArrayList<Integer> GetLeastNumbers\_Solution(int [] input, int k) {

ArrayList<Integer> result = new ArrayList<Integer>();

if(k > input.length) return result;

for(int i = 0; i < k ; i ++){//只排前k次

heapSort(input,i,input.length);//进行第i次排序

result.add(input[i]);

}

return result;

}

private void heapSort( int [] input, int root, int end){//小顶堆实现

for(int j = end -1; j >= root; j --){

int parent = (j + root -1)/2;//算出j节点的双亲节点的序号

if(input[parent] > input[j]){//双亲节点大于当前节点，交换位置。

int temp = input[j];

input[j] = input[parent];

input[parent] = temp;

}

}

}

public static void main(String[] args) {

int [] str={4,5,1,6,2,7,3,8};

Solution32 s=new Solution32();

System.out.print(s.GetLeastNumbers\_Solution(str,4));

}

}

分析:

设树深度为k，http://my.csdn.net/uploads/201207/18/1342597015_2320.jpg。从根到叶的筛选，元素比较次数至多2(k-1)次，交换记录至多k 次。所以，在建好堆后，排序过程中的筛选次数不超过下式：

http://my.csdn.net/uploads/201207/18/1342597082_9043.jpg

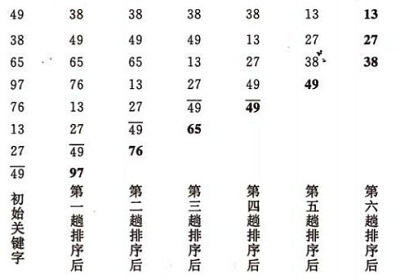
而建堆时的比较次数不超过4n 次，因此堆排序最坏情况下，时间复杂度也为：O(nlogn )。

**7 交换排序—冒泡排序（Bubble Sort）**

@基本思想：

在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。

@冒泡排序的示例：



@算法的实现：

void bubbleSort(int a[], int n){

for(int i =0 ; i< n-1; ++i) {

for(int j = 0; j < n-i-1; ++j) {

if(a[j] > a[j+1])

{

int tmp = a[j] ; a[j] = a[j+1] ; a[j+1] = tmp;

}

}

}

}

对冒泡排序常见的改进方法是加入一标志性变量exchange，用于标志某一趟排序过程中是否有数据交换，如果进行某一趟排序时并没有进行数据交换，则说明数据已经按要求排列好，可立即结束排序，避免不必要的比较过程。本文再提供以下两种改进算法：

（1）设置一标志性变量pos,用于记录每趟排序中最后一次进行交换的位置。由于pos位置之后的记录均已交换到位,故在进行下一趟排序时只要扫描到pos位置即可。

改进后算法如下:

void Bubble\_1 ( int r[], int n) {

int i= n -1; //初始时,最后位置保持不变

while ( i> 0) {

int pos= 0; //每趟开始时,无记录交换

for (int j= 0; j< i; j++)

if (r[j]> r[j+1]) {

pos= j; //记录交换的位置

int tmp = r[j]; r[j]=r[j+1];r[j+1]=tmp;

}

i= pos; //为下一趟排序作准备

}

}

（2）传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值,我们考虑利用在每趟排序中进行正向和反向两遍冒泡的方法一次可以得到两个最终值(最大者和最小者) , 从而使排序趟数几乎减少了一半。

改进后的算法实现为:

void Bubble\_2 ( int r[], int n){

int low = 0;

int high= n -1; //设置变量的初始值

int tmp,j;

while (low < high) {

for (j= low; j< high; ++j) //正向冒泡,找到最大者

if (r[j]> r[j+1]) {

tmp = r[j]; r[j]=r[j+1];r[j+1]=tmp;

}

--high; //修改high值, 前移一位

for ( j=high; j>low; --j) //反向冒泡,找到最小者

if (r[j]<r[j-1]) {

tmp = r[j]; r[j]=r[j-1];r[j-1]=tmp;

}

++low; //修改low值,后移一位

}

}

**8 交换排序—快速排序（Quick Sort）**

@基本思想：

1）选择一个基准元素,通常选择第一个元素或者最后一个元素,

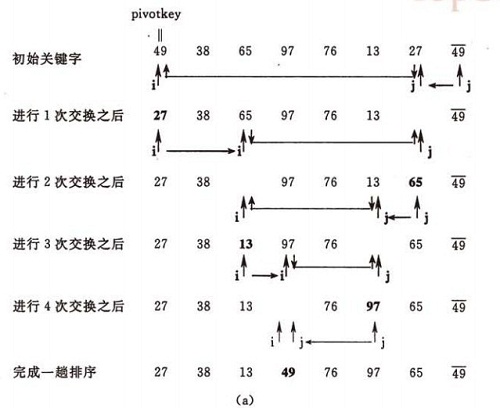
2）通过一趟排序讲待排序的记录分割成独立的两部分，其中一部分记录的元素值均比基准元素值小。另一部分记录的 元素值比基准值大。

3）此时基准元素在其排好序后的正确位置

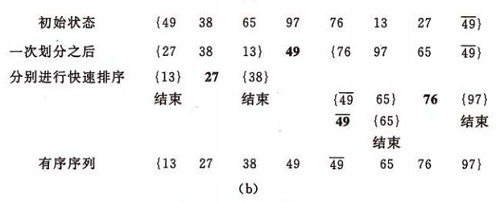
4）然后分别对这两部分记录用同样的方法继续进行排序，直到整个序列有序。

@快速排序的示例：

（a）一趟排序的过程：



（b）排序的全过程



@算法的实现：

package com;

/\*

\* Java实现快速排序算法

\* author:wyr

\* 2016-7-14

\*/

public class QuickSort {

public static void main(String[] args) {

int a[] = {3,1,5,7,2,4,9,6,10,8};

QuickSort obj=new QuickSort();

System.out.println("初始值：");

obj.print(a);

int h=a.length-1;

obj.quickSort(a,0,h);

System.out.println("\n排序后：");

obj.print(a);

}

private void quickSort(int[] a,int low, int high) {

if(low<high){ //如果不加这个判断递归会无法退出导致堆栈溢出异常

int middle=getMiddle(a,low,high);

quickSort(a, 0, middle-1); //递归对低子表递归排序

quickSort(a, middle + 1, high); //递归对高子表递归排序

}

}

public int getMiddle(int[] a, int low, int high){

int key = a[low];//基准元素，排序中会空出来一个位置

while(low<high){

while(low<high && a[high]>=key){//从high开始找比基准小的，与low换位置

high--;

}

a[low]=a[high];

while(low<high && a[low]<=key){//从low开始找比基准大,放到之前high空出来的位置上

low++;

}

a[high]=a[low];

}

a[low]=key;//此时low=high 是基准元素的位置，也是空出来的那个位置

return low;

}

public void print(int a[]){

for(int i=0;i<a.length;i++){

System.out.print(a[i]+" ");

}

}

}

快速排序是通常被认为在同

**分析：**

**快速排序的改进**

数量级（O(nlog2n)）的排序方法中平均性能最好的。但若初始序列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取基准记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个不稳定的排序方法。

在本改进算法中,只对长度大于k的子序列递归调用快速排序,让原序列基本有序，然后再对整个基本有序序列用插入排序算法排序。实践证明，改进后的算法时间复杂度有所降低，且当k取值为 8 左右时,改进算法的性能最佳。算法思想如下：

void print(int a[], int n){

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

cout<<a[j] <<" ";

}

cout<<endl;

}

void swap(int \*a, int \*b)

{

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

int partition(int a[], int low, int high)

{

int privotKey = a[low]; //基准元素

while(low < high){ //从表的两端交替地向中间扫描

while(low < high && a[high] >= privotKey) --high; //从high 所指位置向前搜索，至多到low+1 位置。将比基准元素小的交换到低端

swap(&a[low], &a[high]);

while(low < high && a[low] <= privotKey ) ++low;

swap(&a[low], &a[high]);

}

print(a,10);

return low;

}

void qsort\_improve(int r[ ],int low,int high, int k){

if( high -low > k ) { //长度大于k时递归, k为指定的数

int pivot = partition(r, low, high); // 调用的Partition算法保持不变

qsort\_improve(r, low, pivot - 1,k);

qsort\_improve(r, pivot + 1, high,k);

}

}

void quickSort(int r[], int n, int k){

qsort\_improve(r,0,n,k);//先调用改进算法Qsort使之基本有序

//再用插入排序对基本有序序列排序

for(int i=1; i<=n;i ++){

int tmp = r[i];

int j=i-1;

while(tmp < r[j]){

r[j+1]=r[j]; j=j-1;

}

r[j+1] = tmp;

}

}

int main(){

int a[10] = {3,1,5,7,2,4,9,6,10,8};

cout<<"初始值：";

print(a,10);

quickSort(a,9,4);

cout<<"结果：";

print(a,10);

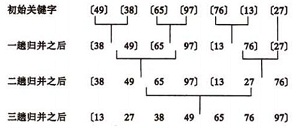
}

**9归并排序（Merge Sort）**

@基本思想：

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

归并排序示例：



@合并方法：

设r[i…n]由两个有序子表r[i…m]和r[m+1…n]组成，两个子表长度分别为n-i +1、n-m。

j=m+1；k=i；i=i; //置两个子表的起始下标及辅助数组的起始下标

若i>m 或j>n，转⑷ //其中一个子表已合并完，比较选取结束

//选取r[i]和r[j]较小的存入辅助数组rf

如果r[i]<r[j]，rf[k]=r[i]； i++； k++； 转⑵

否则，rf[k]=r[j]； j++； k++； 转⑵

//将尚未处理完的子表中元素存入rf

如果i<=m，将r[i…m]存入rf[k…n] //前一子表非空

如果j<=n , 将r[j…n] 存入rf[k…n] //后一子表非空

合并结束。

//将r[i…m]和r[m +1 …n]归并到辅助数组rf[i…n]

void Merge(ElemType \*r,ElemType \*rf, int i, int m, int n)

{

int j,k;

for(j=m+1,k=i; i<=m && j <=n ; ++k){

if(r[j] < r[i]) rf[k] = r[j++];

else rf[k] = r[i++];

}

while(i <= m) rf[k++] = r[i++];

while(j <= n) rf[k++] = r[j++];

}

**10 桶排序/基数排序(Radix Sort)**

说基数排序之前，我们先说桶排序：

基本思想：是将阵列分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递回方式继续使用桶排序进行排序）。桶排序是鸽巢排序的一种归纳结果。当要被排序的阵列内的数值是均匀分配的时候，桶排序使用线性时间（Θ（n））。但桶排序并不是 比较排序，他不受到 O(n log n) 下限的影响。简单来说，就是把数据分组，放在一个个的桶中，然后对每个桶里面的在进行排序。

例如要对大小为[1..1000]范围内的n个整数A[1..n]排序

首先，可以把桶设为大小为10的范围，具体而言，设集合B[1]存储[1..10]的整数，集合B[2]存储 (10..20]的整数，……集合B[i]存储( (i-1)\*10, i\*10]的整数，i = 1,2,..100。总共有 100个桶。

然后，对A[1..n]从头到尾扫描一遍，把每个A[i]放入对应的桶B[j]中。 再对这100个桶中每个桶里的数字排序，这时可用冒泡，选择，乃至快排，一般来说任 何排序法都可以。

最后，依次输出每个桶里面的数字，且每个桶中的数字从小到大输出，这 样就得到所有数字排好序的一个序列了。

假设有n个数字，有m个桶，如果数字是平均分布的，则每个桶里面平均有n/m个数字。如果

对每个桶中的数字采用快速排序，那么整个算法的复杂度是

O(n + m \* n/m\*log(n/m)) = O(n + nlogn - nlogm)

从上式看出，当m接近n的时候，桶排序复杂度接近O(n)

当然，以上复杂度的计算是基于输入的n个数字是平均分布这个假设的。这个假设是很强的 ，实际应用中效果并没有这么好。如果所有的数字都落在同一个桶中，那就退化成一般的排序了。

前面说的几大排序算法 ，大部分时间复杂度都是O（n2），也有部分排序算法时间复杂度是O(nlogn)。而桶式排序却能实现O（n）的时间复杂度。但桶排序的缺点是：

1）首先是空间复杂度比较高，需要的额外开销大。排序有两个数组的空间开销，一个存放待排序数组，一个就是所谓的桶，比如待排序值是从0到m-1，那就需要m个桶，这个桶数组就要至少m个空间。

2）其次待排序的元素都要在一定的范围内等等。

桶式排序是一种分配排序。分配排序的特定是不需要进行关键码的比较，但前提是要知道待排序列的一些具体情况。

分配排序的基本思想：说白了就是进行多次的桶式排序。

基数排序过程无须比较关键字，而是通过“分配”和“收集”过程来实现排序。它们的时间复杂度可达到线性阶：O(n)。

实例:

扑克牌中52 张牌，可按花色和面值分成两个字段，其大小关系为：

花色： 梅花< 方块< 红心< 黑心

面值： 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

若对扑克牌按花色、面值进行升序排序，得到如下序列：

http://my.csdn.net/uploads/201207/21/1342855538_2075.jpg

即两张牌，若花色不同，不论面值怎样，花色低的那张牌小于花色高的，只有在同花色情况下，大小关系才由面值的大小确定。这就是多关键码排序。

为得到排序结果，我们讨论两种排序方法。

方法1：先对花色排序，将其分为4 个组，即梅花组、方块组、红心组、黑心组。再对每个组分别按面值进行排序，最后，将4 个组连接起来即可。

方法2：先按13 个面值给出13 个编号组(2 号，3 号，...，A 号)，将牌按面值依次放入对应的编号组，分成13 堆。再按花色给出4 个编号组(梅花、方块、红心、黑心)，将2号组中牌取出分别放入对应花色组，再将3 号组中牌取出分别放入对应花色组，……，这样，4 个花色组中均按面值有序，然后，将4 个花色组依次连接起来即可。

设n 个元素的待排序列包含d 个关键码{k1，k2，…，kd}，则称序列对关键码{k1，k2，…，kd}有序是指：对于序列中任两个记录r[i]和r[j](1≤i≤j≤n)都满足下列有序关系：

其中k1 称为最主位关键码，kd 称为最次位关键码 。

两种多关键码排序方法：

多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序逐次排序，分两种方法：

最高位优先(Most Significant Digit first)法，简称MSD 法：

1）先按k1 排序分组，将序列分成若干子序列，同一组序列的记录中，关键码k1 相等。

2）再对各组按k2 排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码kd 对各子组排序后。

3）再将各组连接起来，便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法一即是MSD 法。

最低位优先(Least Significant Digit first)法，简称LSD 法：

1) 先从kd 开始排序，再对kd-1进行排序，依次重复，直到按k1排序分组分成最小的子序列后。

2) 最后将各个子序列连接起来，便可得到一个有序的序列, 扑克牌按花色、面值排序中介绍的方法二即是LSD 法。

基于LSD方法的链式基数排序的基本思想

　　“多关键字排序”的思想实现“单关键字排序”。对数字型或字符型的单关键字，可以看作由多个数位或多个字符构成的多关键字，此时可以采用“分配-收集”的方法进行排序，这一过程称作基数排序法，其中每个数字或字符可能的取值个数称为基数。比如，扑克牌的花色基数为4，面值基数为13。在整理扑克牌时，既可以先按花色整理，也可以先按面值整理。按花色整理时，先按红、黑、方、花的顺序分成4摞（分配），再按此顺序再叠放在一起（收集），然后按面值的顺序分成13摞（分配），再按此顺序叠放在一起（收集），如此进行二次分配和收集即可将扑克牌排列有序。

基数排序:

是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

算法实现：

Void RadixSort(Node L[],length,maxradix)

{

int m,n,k,lsp;

k=1;m=1;

int temp[10][length-1];

Empty(temp); //清空临时空间

while(k<maxradix) //遍历所有关键字

{

for(int i=0;i<length;i++) //分配过程

{

if(L[i]<m)

Temp[0][n]=L[i];

else

Lsp=(L[i]/m)%10; //确定关键字

Temp[lsp][n]=L[i];

n++;

}

CollectElement(L,Temp); //收集

n=0;

m=m\*10;

k++;

}

}

**11 排序总结**



## 十、Stack Queue List

**1 Map的遍历**

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

/\*\*

\* Map的遍历，这个遍历比较特殊，有技巧

\*

\* @author leizhimin 2009-7-22 15:15:34

\*/

public class TestMap {

public static void main(String[] args) {

Map<String, String> map = new HashMap<String, String>();

map.put("1" , "a" );

map.put("2" , "b" );

map.put("3" , "c" );

//最简洁、最通用的遍历方式

for (Map.Entry<String, String> entry : map.entrySet()) {

System.out.println(entry.getKey() + " = " + entry.getValue());

}

//Java5之前的比较简洁的便利方式1

System.out.println("----1----" );

for (Iterator<Map.Entry<String, String>> it = map.entrySet().iterator(); it.hasNext();) {

Map.Entry<String, String> entry = it.next();

System.out.println(entry.getKey() + " = " + entry.getValue());

}

//Java5之前的比较简洁的便利方式2

System.out.println("----2----" );

for (Iterator<String> it = map.keySet().iterator(); it.hasNext();) {

String key = it.next();

System.out.println(key + " = " + map.get(key));

}

}

}

3 = c

2 = b

1 = a

----1----

3 = c

2 = b

1 = a

----2----

3 = c

2 = b

1 = a

Process finished with exit code 0

**2、Queue的遍历**

import java.util.Queue;

import java.util.concurrent.LinkedBlockingQueue;

/\*\*

\* 队列的遍历

\*

\* @author leizhimin 2009-7-22 15:05:14

\*/

public class TestQueue {

public static void main(String[] args) {

Queue<Integer> q = new LinkedBlockingQueue<Integer>();

//初始化队列

for (int i = 0; i < 5; i++) {

q.offer(i);

}

System.out.println("-------1-----" );

//集合方式遍历，元素不会被移除

for (Integer x : q) {

System.out.println(x);

}

System.out.println("-------2-----" );

//队列方式遍历，元素逐个被移除

while (q.peek() != null ) {

System.out.println(q.poll());

}

}

}

-------1-----

0

1

2

3

4

-------2-----

0

1

2

3

4

Process finished with exit code 0

**3 Stack的遍历**

import java.util.Stack;

/\*\*

\* 栈的遍历

\*

\* @author leizhimin 2009-7-22 14:55:20

\*/

public class TestStack {

public static void main(String[] args) {

Stack<Integer> s = new Stack<Integer>();

for (int i = 0; i < 10; i++) {

s.push(i);

}

//集合遍历方式

for (Integer x : s) {

System.out.println(x);

}

System.out.println("------1-----" );

//栈弹出遍历方式

// while (s.peek()!=null) { //不健壮的判断方式，容易抛异常，正确写法是下面的

while (!s.empty()) {

System.out.println(s.pop());

}

System.out.println("------2-----" );

//错误的遍历方式

// for (Integer x : s) {

// System.out.println(s.pop());

// }

}

}

0

1

2

3

4

------1-----

4

3

2

1

0

------2-----

Process finished with exit code 0