**面试问题-互联网企业**

**（加粗为不止考一次）**

**一、数据结构**

* **快速排序及其伪代码（Partition函数）**
* 不同排序算法的时间/空间复杂度（平均、最好、最坏）
* **KMP算法**
* 有向无环图的遍历
* 二叉搜索树
* 完全二叉树
* 平衡二叉树
* **红黑树：**
* **B/B+/B-树：**
* 链表的插入和删除结点
* **队列/栈的区别**
* 哈希表
* 不平衡二叉树如何转化为平衡二叉树
* 树的前、中、后序遍历，给出两个能否推断第三个

**二、操作系统**

* 进程不同生命状态的转换条件
* 操作系统如何进行进程/线程的切换
* 进程/线程通信的几种方式
* **线程和进程的区别**
* 作业调度算法
* **用户级/内核级线程**
* **互斥锁的建立**
* chmod命令详细
* **Linux系统中进程的相关命令**
* 死锁的机制条件（生产者与消费者模式）、预防（银行家算法）和解决
* 操作系统内存管理
* 几种页面置换策略
* 几种进程调度的策略
* 缺页中断怎么办
* 如何解决高并发

**三、计算机网络**

* **DNS协议（域名解析过程）**
* **TCP/IP协议**
* 三次握手协议/四次握手协议（在哪一层次）
* 可靠交付
* 滑动窗口的定义
* UDP协议
* **HTTP协议**
* 客户端、服务器端
* Ping使用的协议、所在工作层
* 数据容灾
* 负载均衡

**四、数据库**

* Map-Reduce原理
* Hive SQL与MySQL区别
* Hadoop集群架构
* 事务是什么
* 数据仓库
* 关系数据库四大特征（ACID）
* **数据库索引建立的优缺点**
* 索引为什么用B+树（与二叉树索引的性能区别）
* 索引（包括分类及优化方式，失效条件，底层结构）
* 内连接与外连接
* 表连接有哪几种方式
* 在实践中如何优化MySQL
* 查询日志
* 什么情况下设置了索引但无法使用
* **如何设置一个高并发的系统**
* 数据库有哪些引擎
* 数据库如何实现回滚
* **Spark原理**

**五、算法题**

* 动态规划
* 贪心算法
* **链表中寻找环**
* **两个链表是否相交**
* **反转链表/树【迭代/递归】**
* 判断两棵树相等（或互为镜像）
* **（两个）栈实现队列（伪代码）**
* 如何对一堆数列“1, 2, 1, 2, 3, 2, 1, 2, 5”按照其**出现次数**进行**排序**，如上述数列中数字1出现了3次排到第一位。
* 如何得到一**字符串数组**的**最长公共后缀**？如“aabb, abcd, aacd, adsf, sdfasff”，**最长公共后缀**为“**cd**”
* 字符串**按单词**逆序
* **如何找到两单链表交叉点**

数据库

**一、Hadoop技术族**

* + **Hive：**数据仓库工具，将结构化数据文件映射为库表，提供类SQL查询功能；
  + **HBase**：分布式、面向列的数据库，适用于非结构化海量数据存储的数据库；
  + **MapReduce**：分布式编程模型和软件框架，用于在集群上编写对海量数据处理的并行化程序；

**二、基于Hadoop的大数据处理框架**

* + 平台管理层
  + **数据分析层：**Hive，提供数据分析工具；
  + **编程模型层：**MapReduce，为大规模数据处理提供一个抽象的并行计算编程模型， 为此模型提供可实施的运行环境；
  + **数据存储层：**HBase，提供分布式、可扩展的大量数据表的存储和管理能力；
  + **文件存储层：**HDFS，利用分布式文件系统技术，通过统一的接口向上层提供对象级文件访问服务能力；

**三、MapReduce的基本架构与运行机构**

* + MR程序：启动作业；
  + JobTracker：协调作业的执行；将MR任务分配给空闲的TaskTracker节点；
  + JobClient：基于接口库编写的客户端程序，从JobTracker获取作业ID，提交MR作业；
  + TaskTracker：任务控制；从HDFS取出数据存入本地磁盘；启动一个TaskRunner准备运行任务；
  + TaskRunner：在新的Java虚拟机中创建MTask或RTask进行运算；
  + MTask、Rtask：具体执行MR的任务；

四、面试常见问题

**1. 数据库和数据仓库的区别**？

**答：**

① 数据库：传统的关系型数据库的主要应用，主要是基本的、日常的事务处理；

② 数据仓库：数据仓库系统的主要应用主要是OLAP（On-Line Analytical Processing），支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

2. **HiveSQL和MySQL的区别？**

**答：**

**① 数据存储位置：**Hive 是建立在 Hadoop 之上的，所有 Hive 的数据都是存储在 HDFS 中的。而数据库则可以将数据保存在块设备或者本地文件系统中。

**② 数据格式：**Hive本身没有设定数据格式，在加载的过程中不会对数据本身进行任何修改，而只是将数据内容复制或者移动到相应的HDFS目录中。而在数据库中，不同的数据库有不同的存储引擎，定义了自己的数据格式。所有数据都会按照一定的组织存储，因此，数据库加载数据的过程会比较耗时。

**③ 数据更新：**Hive中不支持对数据的改写和添加，所有的数据都是在加载的时候中确定好的。而数据库中的数据通常是需要经常进行修改的，因此可以使添加和修改数据。

**④ 索引：**Hive要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于MapReduce的引入，Hive可以并行访问数据，因此即使没有索引，对于[大数据](http://lib.csdn.net/base/hadoop)量的访问，Hive仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了Hive不适合在线数据查询。

**⑤ 执行：**Hive中大多数查询的执行是通过Hadoop提供的MapReduce来实现的。而数据库通常有自己的执行引擎。

**⑥ 数据规模：**由于Hive建立在集群上并可以利用MapReduce进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小。

**3. 实践中如何优化MySQL？**

**答：**以下四条从效果上第一条影响最大，后面越来越小。

① SQL语句及索引的优化

② 数据库表结构的优化

③ 系统配置的优化

④ 硬件的优化

4. **关系数据库的四大特征**：

**答**：

① **原子性**：**原子性属性用于标识事务是否完全地完成**，一个事务的任何更新要在系统上完全完成，如果由于某种原因出错，事务不能完成它的全部任务，系统将返回到事务开始前的状态。

② **一致性**：事务在系统完整性中实施一致性，这通过**保证系统的任何事务最后都处于有效状态**来实现。如果事务成功地完成，那么系统中所有变化将正确地应用，系统处于有效状态。**如果在事务中出现错误，那么系统中的所有变化将自动地回滚，系统返回到原始状态**。因为事务开始时系统处于一致状态，所以现在系统仍然处于一致状态。

③. **隔离性**：在隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保**每一事务在系统中认为只有该事务在使用系统**。

④. **持久性**：持久性意味着一旦事务执行成功，在系统中产生的所有变化将是永久的。应该存在一些检查点防止在系统失败时丢失信息。甚至硬件本身失败，系统的状态仍能通过在日志中记录事务完成的任务进行重建。**持久性的概念允许开发者认为不管系统以后发生了什么变化，完成的事务是系统永久的部分**。

5.  **数据库建立索引的优缺点**：·

**答**：

**优点：**

① 通过创建唯一性索引，可以保证[数据库](http://lib.csdn.net/base/mysql)表中每一行数据的唯一性。

② 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

③ 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

④ 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

⑤ 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

**缺点：**

① 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

② 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间。如果要建立聚簇索引，那么需要的空间就会更大。

③ 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

**6. 什么情况下设置了索引但无法使用？**

**答：**

① 以“%”开头的LIKE语句，模糊匹配

② OR语句前后没有同时使用索引

③ 数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）

**7. 事务是什么？**

**答：**数据库事务是指作为**单个逻辑工作单元执行的一系列操作**，要么完全地执行，要么完全地不执行。事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源

**8. 索引的底层实现原理和优化？**

**答：**B+树，经过优化的B+树；主要是在所有的叶子结点中增加了指向下一个叶子节点的指针，因此InnoDB建议为大部分表使用默认自增的主键作为主索引。

**9. 如何设计一个高并发的系统？**

**答：**

① 数据库的优化，包括合理的事务隔离级别、SQL语句优化、索引的优化

② 使用缓存，尽量减少数据库的IO

③ 分布式数据库、分布式缓存

④ 服务器的负载均衡

**10. 表连接有哪几种方式？**

**答**：

① left OUTER join (左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录

② right OUTER join (右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录

③ （inner） join ([等值连接](https://www.baidu.com/s?wd=%E7%AD%89%E5%80%BC%E8%BF%9E%E6%8E%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YYmWIhuhf4ujfvPWFWuW630ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHDLn1c4n1D1)) 只返回两个表中联结字段相等的行

# 操作系统

**一、进程不同生命状态及其转换条件**

1. **进程的3种基本状态**：进程在运行中不断改变其运行状态，一个进程通常有以下3种状态：

（1）**就绪状态**：当进程已经分配到除了CPU之外的所有必要资源，只要活得处理机便可以执行，这时的进程状态称为就绪状态；

（2）**执行状态**：当进程已经获得处理机，其程序正在处理机上执行时，此时的进程状态称为执行状态；

（3）**阻塞状态**：正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态，引起进程阻塞的事件有许多种，例如：等待I/O完成、申请缓冲区而不能满足、等待信号等；

2. **进程3种状态之间的转换**：

（1）**就绪→执行**：处于就绪状态的进程，当进程调度程序为之分配了处理机后，进程便由就绪状态变为执行状态；

（2）**执行→就绪**：处于执行状态的进程在其执行过程中，因为分配给它的一个时间片已用完而不得不让出处理机，于是由执行状态变为就绪状态；

（3）**执行→阻塞**：正在执行的进程因等待某种事件而无法继续执行时，便从执行状态变为阻塞状态；

（4）**阻塞→就绪**：处于阻塞状态的进程，若其等待的时间已经发生，于是进程由阻塞状态变为就绪状态；

**二、进程**

**（一）进程的定义与性质**

1. **进程映像（进程实体）**：进程控制块PCB（参与并发执行的程序控制数据结构）、程序段、相关数据段，共同构成进程；

2. **进程的创建/撤销**：即PCB的创建/撤销；

3. **进程的特点**：进程是具有独立功能的程序在数据集上的运行过程，是系统进行资源分配和调度的一个独立单位；

4. **进程的创建**：进程在被创建，尚未到达就绪状态；先申请空白PCB并填写控制和管理进程的信息，然后由系统为该进程分配运行时的必要资源，再到达就绪状态；

5. **进程的结束**：系统先将进程置为结束状态，再进一步处理资源释放和回收工作；

**（二）进程的切换**

1. **进程切换的定义**：指当前运行的进程被转换到其他状态后，再回到运行继续执行的过程；

2. **进程切换的步骤**：

（1）**保存**处理器上下文：包括程序计数器和其他寄存信息；

（2）**更新**PCB信息；

（3）把进程PCB**移入**就绪队列；

（4）**选择**另一个进程**执行并更新**其PCB；

（5）**恢复**处理器上下文；

3. **进程切换的必要条件**：

**不能**进行进程的调度和切换的情况有：

（1）在处理中断的过程中；

（2）进程在操作系统内核程序临界区中：必须加锁以防止其他并行程序进入，解锁前不应该切换到其他进程；

（3）在其他需要完全屏蔽中断的原子操作过程中；

**应当**进行进程的调度和切换的情况有：

（1）当发生引起调度条件且当前进程无法继续运行下去时；

（2）当中断处理结束后或自陷处理结束后，返回被中断进程的用户态程序执行现场前；

**（三）进程的通信——进程之间数据交换**

1. **低级通信方式**：PV操作

（1）**P操作：**wait(s)表示进程请求一个资源；

（2）**V操作：**signal(s)表示系统释放一个资源；

2. **高级通信方式**：

（1）**共享存储**：在通信的进程之间存在一块可以直接访问的共享空间，通过这块共享空间的读写操作实现进程之间的信息交换，需要使用互斥同步工具；

（2）**消息传递**：以格式化的消息message为单位；

（3）**管道通信**：管道就是用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件。发送进程向管道提供输入数据，接受进程接收管道输出的数据；

（四）作业调度的层次

一个作业从提交到完成，需要经历三级调度：

1. **作业调度（高级调度）**：按一定的调度原则，从外存上处于后备状态的作业中挑选作业，为其分配内存、I/O设备等必要资源，并建立相应的进程，以使它们获得竞争处理器的权利；

2. **内存调度（中级调度）**：为了提高内存利用率和吞吐率，应当使那些暂时不能运行的进程调至外存等待，把此时的进程状态称为挂起状态；当挂起状态的进程具备了运行条件且内存空闲时，由中级调度算法决定把外存上已具备运行条件的进程重新调入内存，修改状态为就绪状态，让其在就绪队列中等待；

3. **进程调度（低级调度）**：按照某种方法和策略从就绪队列中选取进程，将处理器分配给它们；这是最基本、频率最高的调度；

**三、线程**

**（一）线程的定义与性质**

1. **线程的定义**：线程是一个轻量的进程，是CPU的一个最基本的执行单元，也是程序执行的最小单元；

2. **线程的组成**：线程ID，程序计数器，寄存器，堆栈；

**（二）线程与进程的对比**

1. **调度**：线程是进程中的一个实体，是被系统调度和CPU分派的基本单位；而进程是拥有CPU以外的系统资源的基本单位；

2. **拥有资源**：进程拥有系统资源，而线程只拥有一点点必要的资源；

3. **系统开销：**线程开销极小；

4. **切换**：线程之间的切换未必引发进程的切换；而进程的切换必然引发线程的切换；

**（三）进程的调度**

1. **调度的基本概念**：线程的数量大于处理器个数时，会发生争用现象；处理器的调度就是对处理器进行分配，就是从就绪队列中按照一定的算法，选择一个进程并将处理器分配给它运行，实现处理器的并发执行；

2. **调度的意义**：处理器调度是多道程序操作系统的基础；

3. **调度的方式**：

（1）非剥夺调度方式：当一个进程正在处理机上执行时，即使有更重要或紧迫的进程进入就绪状态，仍然让正在执行的进程继续执行，直到该进程完成或发生某种事件而进入阻塞状态；

（2）剥夺调度方式：若有某个更为重要或更紧急的进程需要使用处理器，则立即暂停正在执行的进程，将处理器分配给该进程；

（3）剥夺的原则：优先权原则、短进程优先原则、时间片原则；

**4. 调度的基本准则：**

（1）CPU利用率：应尽可能让CPU保持在忙碌状态；

（2）系统吞吐量：单位时间内CPU完成的作业数量越高越好；长作业需要消耗较长的CPU时间，会降低系统的吞吐量；

（3）周转时间：作业提交到作业完成的经历时间，包括作业等待、就绪队列中排队、在CPU上运行、I/O操作所花费的时间等；

（4）等待时间：进程处于等待CPU的状态的时间之和；

（5）响应时间：用户从提交请求到首次收到系统产生响应所花费的时间；

**5. 调度的算法：**

|  |  |  |  |
| --- | --- | --- | --- |
| **调度算法名** | **算法描述** | **优点** | **缺点** |
| FIFS  先来先服务算法 |  | 简单，有利于长作业；  有利于CPU繁忙 | 效率低，不利于短作业；  不利于IO繁忙型作业； |
| SJF  短作业优先算法 | 对短作业优先调度； | 平均等待时间、  平均周转时间最短； | 长作业容易饥饿；  未考虑作业紧迫程度； |
| 高响应比优先算法 | 主要用于作业调度； | 同时考虑每个作业的等待  时间与估计运行时间； |  |
| 时间片轮转算法 | 主要用于分时系统； |  |  |

**6. 进程同步：**

（1）基本概念：为了协调进程之间的相互制约关系，达到资源共享，避免进程冲突，引入了进程同步的概念；

（2）临界资源：每次只允许一个进程使用的资源为临界资源；访问前需要先检查可否进入临界区；

（3）进程同步：进程之间因为需要在某些位置上协调工作次序而等待、传递信息所产生的制约关系；

（4）进程互斥：当一个进程进入临界区使用临界资源时，另一个进程必须等待，当占用临界资源的进程退出临界区后，另一个进程才允许去访问此临界资源；

**四、内核级与用户级进程（取决于线程控制块的位置）：**

**（一）内核级线程：**

**内核支持线程**是在**核心空间**实现的；**内核为每个线程在核心空间中设置了一个线程控制块**，用来登记该线程的线程标识符、寄存器值、状态、优先级等信息；所**有对线程的操作**，如创建、撤消和切换等，**都是通过系统功能调用由内核中的相应处理程序完成**；设置了内核支持线程的系统，其**调度是以线程为单位进行的**。

1. **优点：**

在多处理器系统中，**内核能够同时调度同一进程中多个线程并行执行到多个处理器中**；如果进程中的一个线程被阻塞，内核可以调度**同一个进程中的另一个线程**；内核支持线程具有很小的[**数据结构**](http://lib.csdn.net/base/datastructure)和堆栈，**线程的切换比较快，切换开销小**；内核本身也可以使用多线程的方式来实现。

1. **缺点：**

即使CPU**在同一个进程的多个线程之间切换**，也需要**陷入内核**，因此**其速度和效率不如用户级线程**。

**（二）用户级线程:**

用户级线程仅存在于用户空间中，与内核无关；就内核而言，它只是管理常规的进程—单线程进程，而感知不到用户级线程的存在；**每个线程控制块都设置在用户空间中**，所有对线程的操作也在用户空间中由线程库中的函数完成，无需内核的帮助；设置了用户级线程的系统，其**调度**仍是**以进程为单位进行**的。

1. **优点：**

**线程的切换无需陷入内核，故切换开销小，速度非常快；**线程库对用户线程的调度[**算法**](http://lib.csdn.net/base/datastructure)与OS的调度算法无关，因此，线程库可提供多种调度算法供应用程序选择使用；用户级线程的实现与操作系统平台无关。

1. **缺点：**

系统调用的阻塞问题：对应用程序来讲，**一个线程的阻塞将导致整个进程中所有线程的阻塞；多线程应用无法享用多处理机系统中多个处理器带来的好处**。

**（三）组合方式：**

内核支持多KST线程的建立、调度和管理，同时，也允许用户应用程序建立、调度和管理用户级线程；一些内核支持线程对应多个用户级线程，程序员可按应用需要和机器配置对内核支持线程数目进行调整，以达到较好的效果。

优点：

同一个进程内的多个线程可以同时在多处理器上并行执行；在阻塞一个线程时，并不需要将整个进程阻塞。

**（四）两者比较：**

1. **内核支持**

用户级线程可在一个不支持线程的OS中实现；内核支持线程则需要得到OS内核的支持。

1. **处理器分配**

在多处理机环境下，对用户级线程而言主，内核一次只为一个进程分配一个处理器，进程无法享用多处理机带来的好处；

在设置有内核支持线程时，内核可调度一个应用中的多个线程同时在多个处理器上并行运行，提高程序的执行速度和效率。

1. **调度和线程执行时间**

设置有内核支持线程的系统，其调度方式和算法与进程的调度十分相似，只不过调度单位是线程；

对只设置了用户级线程的系统，调度的单位仍为进程。

因此，在条件相同的情况下，内核支持的线程通常比用户级线程得到更多的CPU执行时间。

1. **切换速度**

用户级线程的切换，通常发生在一个应用程序的诸线程之间，而不需要陷入内核，而且切换的规则也很简单，切换速度比内核支持线程至少快一个数量级。

1. **系统调用**

在典型OS中，**许多系统调用都会引起阻塞。当一个用户级线程执行这些系统调用时，被阻塞的将是整个进**程；当一个内核支持线程执行这些系统调用时，内核只阻塞这个线程，但仍可调度其所属进程的其他线程执行。

**五、堆和栈的区别**

**（一）堆栈空间分配区别：**

1、栈：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈；

2、堆： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。

**（二）堆栈缓存方式区别：**

1、栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放；

2、堆是存放在二级缓存中，生命周期由虚拟机的[垃圾回收](https://www.baidu.com/s?wd=%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YzPW0YuW79nWPBPjP9mWI-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD)算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

**（三）堆栈数据结构区别：**

堆（数据结构）：堆可以被看成是一棵树，如：堆排序；

栈（数据结构）：一种先进后出的数据结构。

算法

**//【单链表翻转】**

static Link **ReverseLink1**(Link head){ //递归方法

if (head.Next==null**||**head.Next.Next==null) //如果链表元素≤1

return head;

Link rHead=**ReverseLink1**(head.Next);

head.Next.Next=head;

head.Next=null;

return rHead;

}

static Link ReverseLink2(Link head){ //遍历方法

Link curr = head.Next;

Link next = null;

Link nextnext = null;

if (curr == null || curr.Next == null) //如果链表元素≤1

{

return head;

}

while (curr.Next != null) {

next = curr.Next; //1

nextnext = next.Next; //2

next.Next = head.Next; //3

head.Next = next; //4

curr.Next = nextnext; //5

}

return head;

}

**//【判断链表中是否有环】**

static boolean **LinkCircle**(Link head){

Link first=head;

Link second=head;

while (second.Next!=null && second.Next.Next!=null){

second=second.Next.Next;

first=first.Next;

if (second==first){

return true;

}

}

return false;

}

**//【判断2个链表是否相交】**

static boolean **IntersectLink**(Link head1, Link head2){

Link first=head1;

Link second=head2;

while (first.Next!=null){

first=first.Next;

}

while (second.Next!=null){

second=second.Next;

}

if(first!=second){

return false;

}

else{

return true;

}

}

**//【用两个栈实现队列】**

public class **Solution** {

Stack<Integer> stack1 = new Stack<Integer>();

Stack<Integer> stack2 = new Stack<Integer>();

public void **push**(int node){

        while(!stack2.isEmpty()){

                stack1.add(stack2.pop());

        }

        stack1.push(node);

}

public int **pop**(){

        while(!stack1.isEmpty()){

                stack2.add(stack1.pop());

        }

        return stack2.pop();

}

}

**//【二叉树镜像】**

static void **MirrorTree**(TreeNode root){

if (root==null){

return ;

}

else{

TreeNode temp=root.Left;

root.Left=root.Right;

root.Right=temp;

**MirrorTree**(root.Left);

**MirrorTree**(root.Right);

}

}