**2021**年百度精选面试题及答案

1. 输入www.baidu.com在浏览器的完整过程，越详细越好。

浏览器获取输入的域名ww. baidu. com

浏览器向域名系统DNS请求解析ww. baidu. com的IP地址

DNS解析出百度服务器的IP地址

浏览器与服务器建立TCP连接（默认端口 80）

浏览器发出HTTP请求，请求百度首页

服务器通过HTTP请求把首页文件发给浏览器

TCP连接释放

浏览器解析首页文件*，*展示web界面

1. 请描述C/C++程序的内存分区？

其实c和c卄的内存分区还是有一定区别的，但此处不作区分：

1） 、栈区（stack）-由编译器自动分配軽放,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） 、堆区（heap） ——般由程序员分配释放，若程序员不軽放，程序结束时可能由 OS回

收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3） 、全局区（静态区）（static） —，全局变量和静态变量的存储是放在一块的，初 始化的

全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻 的另

—块区域。-程序结束后由系统释放。

4） 、文字常量区一常量字符串就是放在这里的。程序结束后由系统释放。

5） 、程序代码区一存放函数体的二进制代码。

栈区与堆区的区别：

1） 堆和栈中的存储内容：栈存局部变量•,函数参数等。堆存储使用new、malloc申请 的变量等；

2） 申请方式：栈内存由系统分配，堆内有由自己申请；

3） 申请后系统的响应：栈一一只要栈的剩余空间大于所申请空间，系统将为程序提供 内存，否则将报异常提示栈溢出。

堆一一首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请 时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结 点链表中删除，并将该结点的空间分配蜘程序；

1. 申请大小的限制：WindowsT栈的大小一般是2M,堆的容量较大；
2. 申请效率的比较：栈由系统自动分配，速度较快。堆使用new、malloc等分配，较 慢；

总结：栈区优势在处理效率，堆区优势在于灵活；

内存模型：自由区、静态区、动态区；

根据c/c++对象生命周期不同，c/c++的内存模型有三种不同的内存区域，即：自由存 储区，动态区、静态区。

自由存储区：局部非静态变量的存储区域，即平常所说的栈；

动态区：用new , malloc分配的内存，即平常所说的堆；

靜态区：全局变量，静态变量，字符串常量存在的位置；

注：代码虽然占内存，但不属于c/c卄内存模型的一部分；

1. 快速排序的思想、时间复杂度、实现以及优化方法？

快速排序的三个步骤

1. 选择基准：在待排序列中，按照某种方式挑出一个元素，作为’基准(pivot);
2. 分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准 左边的元素都比该基准小，在基准右边的元素都比基准大：
3. 通归地对两个序列进行快速排序，直到序列为空或者只有一个元素。

基准的选择：

对于分治算法，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效 率会达到最大。

艮卩：同一数组，时间复杂度最小的是每次选取的基准都可以将序列分为两个等长的；时 间复杂度最大的是每次选择的基准都是当前序列的最大或最小元素；

快排代码实现：

我们一般选择序列的第一个作为基数，那么快排代码如下：

void quicksort(vector<int> int left, irrt right)

[

if (left < right)//false 则递归结束

{

int key=v[left];//基数賦值

int low = left:

int high = right:

while (low < high) //当 lou«=high 时，表示一抡分割结束

{

while (low < high && v [high] >= key)〃v[low]为基数，从后向前与基数比 较

!

high—:

}

swzp (v[lov/], v [high]): while (low < high && v[low] <= key)//v [high] ^1 基数，从前向后与基数比 较

!

lovH-；

}

(vflow]3 v[high]):

}

//分割后，对每一分段重复上述操作

quicksort (v, left, low-1):

quicksort (v, lovrt-1, right):

注：上述数组或序列V必须是引用类型的形参，因为后续快排结果需要直接反映在原序 列中；

优化：

上述快排的基数是序列的第一个元素，这样的对于有序序列，快排时间复杂度会达到最 差的0(n\*2)o所以，优化方向就是合理的选择基数。

常见的做法“三数取中〃法(序列太短还要结合其他排序法，如插入排序、选择排序等), 如下：

1. 当序列区间长度小于7时，采用插入排序：
2. 当序列区间长度小于40时，将区间分成2段，得到左端点、右端点和中点，我们对 这三个点取中数作为基数；
3. 当序列区间大于等于40时，将区间分成8段，得到左三点、中三点和右三点，分 别再得到左三点中的中数、中三点中的中数和右三点中的中数，再将得到的三个中数取 中数，然后将该值作为基数。

具体代码只是在上一份的代码中将“基数赋值"改为①②③对应的代码即可：

int key=v[left];//基数谶值

if (right-left+l<=7) {

insertion\_sort (v, left, right);〃插入排序

return;

Jelse if (right-left+l<=8) (

key=SelectPivotOfThree (v, left, right)；〃三个取中

}else{

〃三组三个取中，再三个取中(使用4次SelectPivotOfThree，此处不具体展示)

}

需要调用的函数：

void insertion\_sort (vector<irrt> &unsotrted, irrt left, irrt right) //插入排序算法

[

for (int i = left+1: i <= right: i++)

[

if (unsorted[i - 1] > unsorted[i])

int temp = unsorted[i]: int j = i;

while (j > left && unsorted[j - 1] > temp) [

unsorted[j] = unsorted[j - 1]:

unsorted [j] = tenp:

int SelectPivotOf Three (vector<int> &arr, int low, irrt high)

//三数取中，同时将中值移到序列第一位

[

int mid = low + (high - low)/2;〃计算数组中间的元素的下标

//使用三数取中法迭择枢轴

if (arr[mid] > arr[high])//目标：arr[mid] <= arr[high]

[

swzp (arr [mid], arr [high]):

}

if (arr[low] > arr[high])//目标：arr[low] <= arr[high]

[

swzp (arr [low], arr [high]):

}

if (arr [mid] > arr[low]) 〃目标：arr[low] >= arr[mid]

[

swzp (arr [mid], arr [low]):

}

〃此时，arr [mid] <= arr[low] <= arr [high]

return arr[low]：

//low的位置上保存这三个位置中间的值

〃分割时可以直接使用low位置的元素作为枢轴，而不用改変分割函数了

}

这里需要注意的有两点：

1. 插入排序算法实现代码；
2. 三数取中函数不仅仅要实现取中，还要将中值移到最低位，从而保证原分割函数依然 可用。
3. 请描述IO多路复用机制?

10模型有4中：同步阻塞10、同步非阻塞10、异步阻塞10、异步非阻塞10； 10多路 复用属于10模型中的异步阻塞10模型，在服务器高性能10构建中常常用到*。*

同步异步是表示服务端的，阻塞非阻塞是表示用户端，所以可解释为什么10多路复用 （异步阻塞）常用于服务器端的原因；

文件描述符（知，又叫文件句柄）：描述符就是一个数字，它指向内核中的一个结构体 （文件路径，数据区等属性）。具体来源：Linux内核将所有外部设备都看作一个文件来 操作，对文件的操作都会调用内核提供的系统命令，返回一个fd（文件描述符）。 下面开始介绍10多路复用：

（1） I/。多路复用技术通过把多个I/。的阻塞复用到同一个select、poll或epoll的 阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线 程/多进程模型比，I/。多路复用的最大优势是系统开销小，系统不需要创建新的额外 进程或者线程。

（2） select, poll, epoll本质上都是同步I/O,因为他们都需要在读写事件就绪后自 己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读 与，异步I/O的实现会负责把数据从内核拷贝到用户空间。

（3） I/O多路复用的主要应用场景如下：

服务器需要同时处理多个处于监听状态或者多个连接状态的套接字；

服务器需要同时处理多种网络协议的套接字；

（4） 目前支持I/O多路复用的系统调用有select, poll, epoll, epoll与select的 原理比较类似，但epoll作了很多重大改进，现总结如下：

1. 支持一个进程打开的文件句柄知个数不受限制（为什么select的句柄数量受限制: select使用位域的方式来传逢关心的文件描述符，因为位域就有最大长度，在Linux 下是1024,所以有数量限制）5
2. I/O效率不会随着FD数目的増加而线性下降；

©epoll的API更加简单；

（5） 三种接口调用介绍：

1. select函数调用格式：

#include <sys/select. h>

#include < sys/1ime. h>

int select（int maxfdpi, fd\_set \*readset, fd\_set \*wxiteset, fd\_set

\*exceptset, const struct timeval \*tineout）

//返回值：就绪描述符的数目，超时返回。，出错返回T

1. poll®数调用格式：

# include <poll. h>

int poll （ struct pollfd \* fds, unsigned int nfds, int timeout）;

©epoll®数格式（操作过程包括三个函数）：

Sinelude〈sys/epoll. h>

int epoll\_create（int size）;

int epoll\_ctl（int epfd, int op, int fd, struct epoll\_event \*event）;

int epoll\_wait（int epfd, struct epoll\_event \* events, int maxevents, int timeout）;

（6） 作用：一定程度上替代多线程/多正程，减少资源占用，保证系统运行的高效率; 更多细节待续……

1. 实践中如何优化MySQL?

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

®系统配置的优化

@硬件的优化

1. 什么情况下设置了索引但无法使用？
2. LIKE语句，模糊匹配
3. 0R语句

®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）

1. SQL语句的优化.
2. 对查询进行优化，应尽量避免全表扫描，首先应考虑在where及order by涉及的 列上建立索引。
3. 应尽量避免在where子句中使用!=或。操作符，否则将引擎放弃使用索引而进行全 表扫描。
4. 应尽量避免在where子句中对字段进行null值判断，否则将导致引擎放弃使用索 引而进行全表扫描，如：

select id from t where num is null

可以在num上设置默认值0,确保表中num列没有null值，然后这样查询：

select id from t where num=0

1. 应尽量避免在where子句中使用or来连接条件，否则将导致引擎放弃使用索引而 进行全表扫描，如：

select id from t where num=10 or nun=20

可以这样查询：

select id from t where num=10

union all

select id from t where num=20

1. 下面的查询也将导致全表扫描：

select id from t where name like

若要提高效率，可以考虑全文检索。

1. in和not in也要慎用，否则会导致全表扫描，如：

select id from t where num in（l,2, 3；

对于连续的数值，能用between就不要用in 了：

select id from t where num between 1 and 3

1. 如果在where子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解 析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选 择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择 的输入项。如下面语句将进行全表扫描：

select id from t where num=@num

可以改为强制查询使用索引：

select id from t with (index (索引名)：where num=@num

1. 应尽量避免在where子句中对字段进行表达式操作，这将导致引擎放弃使用索引而 进行全表扫描。如：

select id from t where num/2=100

应改为：

select id from t where num=100\*2

1. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行 全表扫描。如：

select id from t where substring (nane, 1, 3)=, abcJ --name 以 abc 开头的 id select id from t where datediff (day, ere ate date,J 2005-11-305 )=0—' 2005T1-30’ 生成的id

应改为：

select id from t where name like 'abc%'

select id from t where createdate>=, 2005-11一30‘ and createdate<J 2005T2T'

1. 不要在where子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系 统将可能无法正确使用索引。
2. 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的 第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽 可能的让字段顺序与索引顺序相一致。
3. 不要写一些没有意义的查询，如需要生成一个空表结构：

select coll, col2 into #t from t wheie 1=0

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

create table St (...)

1. 很多时候用exists代替in是一个好的选择：

select num from a where num in(select num from b)

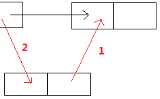
用下面的语句替换：

select num from a where exists (select 1 from b where num=a. num)

1. 并不是所有索引对查询都有效,SQL是根据表中数据来进行查询优化的，当索引列 有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex, male、female 几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。
2. 索引并不是越多越好，索引固然可以提高相应的select的效率，但同时也降低了 insert及update的效率，因为insert或update时有可能会重建索引，所以怎样 建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则 应考虑一些不常使用到的列上建的索引是否有必要。
3. 应尽可能的避免更新clustered索引数据列，因为clustered索引数据列的顺序 就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费 相当大的资源。若应用系统需要频繁更新clustered索引数据列，那么需要考虑是否 应将该索引建为clustered索引。
4. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查 询和连接的性能，并会増加存储开销。这型因为引擎在处理查询和连接时会逐个比较字 符串中每一个字符，而对于数字型而言只需要比较一次就够了。
5. 尽可能的使用varchar/nvarchar代替char/nchar ,因为首先变长字段存储空间 小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要 高些。
6. 任何地方都不要使用select \* from t ,用具体的字段列表代替“\*” ,不要返回 用不到的任何字段。
7. 尽量使用表变量来代替临时表*。*如果表变量包含大量数据*，*请注意索引非常有限（只 有主键索引）。
8. 避免频繁创建和删除临时表，以减少系统表资源的消耗。
9. 临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重 复引用大型表或常用表中的某个数据集旺。但是，对于一次性事件，最好使用导出表。
10. 在新建临时表时，如果一次性插入数据量很大，那么可以使用select into代替 create table,避免造成大量log ,以提高速度；如果数据量不大，为了緩和系统表 的资源，应先create table,然后inserto
11. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ,然后drop table ,这样可以避免系统表的较长时间锁定。
12. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么 就应该考虑改写。
13. 使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题， 基于集的方法通常更有效。
14. 与临时表一样，游标并不是不可使用。对小型数据集使用FAST\_FORWARD游标通常 要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集 中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标 的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。
15. 在所有的存储过程和触发器的开始处设置SET NOCOUOT ON ,在结束时设置SET NOCOUOT OFF。无需在执行存儲过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。
16. 尽量姦免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。
17. 尽量避免大事务操作*，*提高系统并发能力。
18. 在一个带头结点的单链表HL中，若要在第一个元素之前插 入一个由指针p指向的结点。其语句为？

在插入节点时:先要将待插入节点P的后继节点设为第一个元素，也就是p->next=HL >nexto然后再将头结点HL的后继节点改为P节点,HL->next=p。下图中红色的箭头说 明

了插入操作执行的顺序,如果顺序不当，就会丢失指向第一个元素的指针，破坏链表结构



个 HL

个

P

其语句为：P- >nex t=HL- >nex t; HL~ >nex t=p;

1. 如何设计一个高并发的系统？
2. 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;
3. 使用緩存，尽量减少数据库10；

®分布式数据库、分布式緩存；

@服务器的负载均衡；

1. 两条相交的单向链表，如何求他们的第一个公共节点？
2. 如果两个链表相交，则从相交点幵始，后面的节点都相同，即最后一个节点肯定相同;
3. 从头到尾遍历两个链表，并记录链表长度，当二者的尾节点不同，则二者肯定不相交;

①尾节点相同，如果A长为LA, B为LB，如果LA>LB,则A前LA-LB个先跳过；

——更多如链表相关经典问题：求单向扇8循环链表的入、将两个有序链表合并合成一 个有序链表、链表逆序、求倒数第K个节点，判断是否有环等。

1. new/delete 和 malloc/free 的底层实现？

malloc和new的区别：

1. malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都 可用于申请动态内存和释放内存；
2. new返回指定类型的指针，并且可以白动计算所需要大小。而malloc则必须要由 程序员计算字节数，并且在返回后强行转换为实际类型的指针；
3. new/delete在对象创建的同时可以自动执行构造函数初始化，在对象在消亡之前会 自动执行析构函数。而malloc只管分配内存，并不能对所得的内存进行初始化，所以 得到的一片新内存中，其值将是随机的：

既然new/delete的功能覆盖了 malloc/free,为什么C++还要保留malloc/free?因为 CF呈序经常要调用C函数，而C程序只能用malloc/free管理动态内存。

new/delete、malloc/free 底层实现原理：

概述：new/delete的底层实现是调用malloc/free©数实现的，而malloc/free的底 层实现也不是直接操作内存而是调用系统API实现的。

new/delete的两种分配方式原理图如下：

注意，针对上图最末尾所述的"new []/delete []时会多开辟4字节用于存储对象个数〃, 作如下说明：

1. 对于内置类型：

new []不会在首地址前4个字节定义数组长度。

delete和delete []是一样的执行效果，都会删除整个数组，要删除的长度从new时即 可知道。

1. 对于自定义类型：

new []会在首地址前4个字节定义数组长度。

当delete []时，会根据前4个字节所定义的长度来执行析构函数删除整个数组。

如果只是delete数组首地址，只会删除第一个对象的值。

1. overload、override、overwrite 的介绍.

CD overload （重载），即函数重载：

1. 在同一个类中；
2. 函数名字相同；
3. 函数参数不同（类型不同、数量不同，两者淹足其一即可）：

㈤不以返回值类型不同作为函数重载的条件。

（2） override （覆盖，子类改写父类的虚函数），用于实现C卄中多态：

1. 分别位于父类和子类中；
2. 子类改写父类中的virtual方法；

①与父类中的函数原型相同。

（3） overwrite （重写或叫隐藏，子类改写父类的非虚函数，从而屏蔽父类函数）：

1. 与overload类似，但是范围不同，是子类改写父类；
2. 与override类似，但是父类中的方法不是虚函数。
3. 什么是守护进程？

（1） 什么是守护进程？

守护进程（DaemonProcess）,也就是通常说的Daemon进程（精灵进程）, 是Linux 中的后台服务逬程。它是一个生存期较长的逬程，通常独立于 控制终端并且周期性地执行某种任务或等待处理某些发生的事件。

守护进程是个特殊的孤儿进程，这种进程脱离终端，为什么要脱离终端呢？之所以脱离 于终端是为了避免进程被任何终端所产生的信息所打断，其在执

行过程中的信息也不在任何终端上显示。

（2） 如何查看守护进程？

在终端敲：PS axj

从上图可以看出守护进行的一些特点：

守护进程基本上都是以超级用户启动（UID为0 ）

没有控制终端（TIY为？）

终端进程组ID为-1 （ TPGID表示终端进程组ID）

1. 请描述小端/大端机器.

小端/大端的区别是指低位数据存储在内有低位还是高位的区别。其中小端机器指：数 据低位存储在内存地址低位，高位数据则在内存地址高位；大端机器正好相反。

当前绝大部分机器都是小端机器，就是比较符合人们逻辑思维的数据存储方式，比如 intel的机器基本就都是小端机器。

1. 请描述长连接与短连接.

（1） 就是TCP长连接和TCP短连接：

1. "P长连接："P长连接指建立连接后保持连接而不断开。若一段时间内没有数据传 输，服务器会发送心跳包给客户端，判断客户端是否还在线，叫做长连接中的keep alive。一般步骤：连接-数据传输-保持连接（心跳）-数据传输-保持连接（心跳） -……-关闭连接；
2. "P短连接：指连接建立并传输数据完成后，就断开连接。一般步骤：连接-数据传 输-关闭连接；

①使用场景：长连接适合单对单通信且蟻数不太多的情况；短连接适合连接数多且经 常更换连接对象的；

（2） HTTP是什么连接：

1. 在HTIP/1.0中，默认使用的是短连接。但从HTIP/1.1起，默认使用长连接，用以 保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码：

Connection：keep-alive

注意：此处的keep-alive和上述TCP长连接原理介绍中的keep alive不是一个意思: 此处表示告知服务器本http请求是长连接模式，而TCP长连接中的keep alive表示对 客户端的保活检测。

1. http长连接并不是一直保持连接

http的长连接也不会是永久保持连接，它有一个保持时间如20s （从上一；欠数据传输完 成开始计时）*，*可以在不同的服务器软件（如Apache）中设定这个时间，若超过该时 间限制仍然无数据通信传输，服务器就主动关闭该连接。注：实现长连接要客户端和服 务端都支持长连接。

©http连接实质：http的长连接/短连接实质上就是"P的长/短连接。

1. C++中引用与指针的联系与区别？

联系：引用是变量的别名，可以将引用看做操作受限的指针; 区别：

1） 指针是一个实体，而引用仅是个别名；

2） 引用只能在定义时必须初始化，指针可以不初始化为空；

3） 引用初始化之后其地址就不可改变（則始终作该变量的别名直至销毁，即从一而终。 注意:并不表示引用的值不可变,因为只要所指向的变量值改变。引用的值也就改变了）, 但指针所指地址是不可变的；如下：

int m=23, n=13;

int& a=m;

a=12; //合法，相当于修改m=12

a=n; //合法，相当于修改m=13

1. 一个大的含有50M个URL的记录，一个小的含有500个

URL的记录，找岀两个记录里相同的URLo

首先使用包含500个uzl的文件创建一个hash.seto

然后遍历50M的url记录，如果url在hash\_set中,则输出此url并从hash\_set中删 除这个url。

所有输出的url就是两个记录里相同的urlo

1. 海量日志数据，提取岀某日访问百度次数最多的那个IP。

如果日志文件足够的大，大到不能完全加载到内存中的话。

那么可以考虑分而治之的策略，按照IP她址的haSh（IP）%1024值，将海量日志存储到 1024个小文件中。毎个小文件最多包含4M个IP地址。

对于每个小文件，可以构建一个IP作为key,出现次数作为value的hash\_map,并记 录当前出现次数最多的1个IP地址。

有了 1024个小文件中的出现次数最多的IP,我们就可以轻松得到总体上出现次数最多 的IP。

17.有10个文件，每个文件1G,每个文件的每一行都存放 的是用户的query,每个文件的query都可能重复。如何 按照query的频度排序？

1）读取10个文件，按照hash （query） %13的结果将query写到对应的文件中。这样我 们就有了 10个大小约为1G的文件。任意一个query只会出现在某个文件中。

1. 对于1)中获得的1。个文件，分别进行如下操作

-利用hash\_map ( query, query\_count)来统计每个query出现的次数。

-利用堆排序算法对query按照出现次数进行排序。

-将排序好的query输出的文件中。

这样我们就获得了 10个文件，每个文件中都是按频率排序好的query。

1. 对2)中获得的10个文件进行归并排序，并将最终结果输出到文件中。
2. 有一根27厘米长的细木杆，在第3厘米，7厘米，11 厘米，17厘米，23厘米这五个位置上各有一只蚂蚁，木杆 很细，不能同时通过两只蚂蚁，开始时，蚂蚁的头朝向左还 是右是任意的，他们只会朝前走或掉头，但不会后退，当两 只蚂蚁相遇后，蚂蚁会同时掉头朝反方向走，假设蚂蚁们每 秒钟可以走1厘米的距离。求所有蚂蚁都离开木杆的最小时 间和最大时间。

两只蛆蚁相谒后，各自掉头朝相反方向走。如果我们不考虑每个蛆蚁的具体身份，这和 两只蛆蚁相谒后*，*打个招呼继续向前走'没有什么区别*。*

所有蛆蚁都离开木杆的最小时间为

max (min(3, 27-3),min(7, 27-7), min(11,27-11), min(17, 27-17), min(23, 27-23))=11 所有蛆蚁都离开木杆的最大时间为

max (max(3, 27-3), max(7, *27~7),* max (11,27-11), max (17, 27-17), max(23, 27-23))=24

1. 判断两棵树是否相等，请实现两棵树是否相等的比较， 相等返回1,否则返回其他值，并说明算法复杂度。

数据结构为：

typedef struct TreeNode

[

char c;

TreeNode \*leftchild;

TreeNode \*rightchild;

}TreeNode;

函数接口为：int CompTree (TreeNode\* -reel, TreeNode\* tree2);

注：A、B两棵树相等当且仅当RootA->c==RootB->c,而且A和B的左右子树相等或者 左右互换相等。

递归方法:

bool CompTree(TreeNode \*treel, TreeHode \*tree2)

[

if (treel == NULL && tree2 == NULL)

return true;

if (treel == NULL || tree2 == NULL)

return false;

if (treel->c != tree2->c)

return false;

if ( (CompTree (treel->leftchild, tree2->leftchild) &&

CompTree(treel->rightchild, tree2->rightchild)) ||

CompTree(treel->leftchild, -ree2->rightchild) &&

CompTree(treel->rightchild, tree2->leftchild))

return true;

I

时间复杂度：

在树的第。层，有1个节点，我们会进行1次函数调用；

在树的第1层，有2个节点，我们可能会进行4次函数调用；

在树的第2层，有4个节点，我们可能会进行16次函数调用；

在树的第x层，有2“x个节点，我们可能会进行（2\）\*2次函数调用; 所以假设总节点数为n,则算法的复杂度为0（n'2）。

1. 将多个集合合并成没有交集的集合。

给定一个字符串的集合，格式如：{aaabbbccc}, {bbbddd}, {eeefff}, {ggg}, {dddhhh} 要求将其中交集不为空的集合合并，要求合并完成后的集合之间无交集。

例如上例应输出{aaabbbcccdddhhh}, {eeefff}, {ggg} o

（1） 请描述你解决这个问题的思路；

（2） 请给出主要的处理流程，算法，以及算法的复杂度

（3） 请描述可能的改进。

集合使用hash.set来表示，这样合并时间复杂度比较低。

1、 给每个集合编号为。，1, 2, 3...

2、 创建一个hash\_map, key为字符串，value为一个链表，链表节点为字符串所在集 合的编号。遍历所有的集合，将字符串和对应的集合编号插入到hash.map中去。

3、 创建一个长度等于集合个数的int数组，表示集合间的合并关系。例如，下标为5 的元素值为3,表示将下标为5的集合合并到下标为3的集合中去。开始时将所有值都 初始化为-1,表示集合间没有互相合并。在集合合并的过程中，我们将所有的字符串都 合并到编号较小的集合中去。

遍历第二步中生成的hash.map,对于每个value中的链表，首先找到最小的集合编 号（有些集合已经被合并过，需要顺着合并关系数组找到合并后的集合编号）*，*然后将 链表中所有编号的集合都合并到编号最小的集合中（通过更改合并关系数组）。

4、现在合并关系数组中值为-1的集合即为最终的集合，它的元素来源于所有直接或间 接指向它的集合。

算法的复杂度为。（n）,其中n为所有集合中的元素个数。

题目中的例子：

0：{aaabbbccc}

1: {bbbddd}

2：{eeefff}

3： {ggg}

4: {dddhhh}

生成的hash.map,和处理完每个值后的合并关系数组分别为

aaa：0。[T,T,T,T,T]

bbb：0,lo [-1,0,-1,-1,-1]

ccc:0。[-1,0,-1,-1,-1]

ddd:l,4o [-1,0,-1,-1,0]

eee：2。[-1,0,-1,-1,0]

fff：2o [-1,0,-1,-1,0]

ggg：3。[-1,0,-1,-1,0]

hhh:4o [-1,0,-1,-1,0]

所以合并完后有三个集合，第。，1, 4个集合合并到了一起。

1. 平面内有11个点，由它们连成48条不同的直，由这些

点可连成多少个三角形？

首先你要分析，平面中有11个点，如果这些点中任意三点都没有共线的，那么一共应 该有C（ll, 2）=55,可是，题目中说可以连接成48条直线，那么这11个点中必定有 多点共线的情况*。*55-48=7,从7来分析:

假设有一组三个点共线，那么可以组成的直线在55的基础上应该减去C（3, 2）-1=2 2\*3=6尹7,因此，可以断定不仅有三点共线的，也可能有四个点共线的可能。

假设有一组四个点共线，那么可以组成的直线在55的基础上应该减去C （4, 2）-1=5 （备注，五个点共线的可能不存在，因为，C（5, 2）-1=9>7,故不可能有五条直线共线。）

因此，三点共线少2条，4点共线少5条，只有一个4点共线，一个3点共线才能满足 条件，其余情况不能滴足少了7条直线。

那么，这11个点能组成的三角形的个数为,C（11,3）-C（3,3）-0（4,3）=165-1-4=160 （备 注，三个点共线不能组成三角形）

1. 在常用的C编译环境中，已知struct (inta; double b;char

c;}A;求sizeof(A)的返回值？

24

1. 若有宏定义:#define MOD(x, y) x%y则执行以下语句后的

输岀结果是？

int a=13, b=94;

printf(”％d\n”, MOD(b, a+4));

7

1. 用两个栈实现一个FIFO队列.

—个用来入栈，一个用来出栈.

1. 描述IPv6相对IPv4带来了哪些技术差异.

IPv4有4个8位二级制数表示，工4个字节.

浏览器向域名系统DNS请求解析ww. baidu. com的IP地址

IPv6地址空间从IPV4的32位扩展到128位IPv6实现了包头设计的简化，降低了网络 设备对包处理的负荷IPv6实现了实现了她址的自动化配置，无需部署DHCP也可实现地 址配置为了实现IPv6地址解析、路由、囲络控制消息传递等功能，网络需要配合实现邻 居发现协议(Neighbor Discovery) s ICKPVI6、DHCPV6、0SPFV3、BGP4待新协议部 署或扩展IPv6部署过程中，网络可能会部署双栈、隧道或翻译等过渡方案实现与原有 PV4网络互通

1. 内存主要用的4个区是?

栈区，堆区，静态区，代码区

1. Linux下进程通信的方式有哪些？

管道，消息队列，共享内存，信号量，信号，套接口

1. 如何用队列来求一个二叉树的最大高度？

BFS

1. 下列哪个操作可以不需要再内核态执行？

△.系统调用B.malloc/free C.软中断D.内倒硕

B

1. 应用程序ping发岀的是什么报文？

ICMP应答报文

1. 假定对长度为n=119的有序数组进行折半查找，则对应的 判定数的高度是多少？

7

2\*7=128>119

1. 基于数组实现一个环形队列，队列的元素类型为整型，队 列长度128队列提供写入(EnQueue)和读取(Dequeue)两个 接口.写代码定义并实现这个队列的写入和读取接口.要求 队列的写入和读取接口支持多线程并发调用.请在代码中添 加必要注释
2. 队列实现正确给io分

（1） 定义里包含头尾指针（5分）

（2） 写入接口包含判滴逻辑，能够用返回值描述出错情况。（5分）

（3） 读取接口包含判空逻辑。（5分）

1. 读写接口线程安全给1。分

（1） 读逻辑（判断逻辑）需要加锁解锁（5分）

（2） 更新逻辑（指针或者队列长度的更新）需要加锁。（5分）

1. 代码风格糟糕的可适当减分（最多可濾5分）
2. 请描述同一网段下主机Ap主机B的全过程.假定此时主 机A知道主机B的P地址,但不知道主机B的MAC地址
3. 主机A广播姗请求，询问主机B的KA3地址。
4. 主机B收到姗请求,以姗响应形式回复自己的MAC地址。
5. 主机A收到主机B的MAC地址，完成CMP请求报文构造，发送给B。
6. 主机B收到主机A的ICMP请求报文，回复一个ICMP应答报文。
7. 主机A收到CMP应答报文,ping结束。

Ping过程回答正确给5分。姗过程回答正确给5分。

1. 请描述拥有私有地址的局域网内主机向888发送DN查 询请求报文时报文的传输层端口、网络层地址以及数据链路 层地址几个主要字段与其在传输过程中的变化。

主机构造DNS查询请求UDP报文，目标P地址8.8.8.8,源IP地址为私有地址,报文的 政目的地址为局域网网关的MAC地址，源MAC地址为主机网卡MAC地址（对姗过程描 述不做要求），源端口任意，目的端口 532.报文到达网关处,修改源P地址为网关公有IP 地址，同时根据需要修改源端口，然后将报文发送到8.8.8.8o此时一般情况下会移除原 有的MC层头部，根据实际情况有可能会芍其他头部加入。

答出四层细节的给3分

答出三次细节（包括NAT）的给4分

笞出二层细节的给*3*分（不要求回笞逬入外网后的二层情况）

1. 同一主机上有两个进程，其中各有一个变量a和b。请问

a和b的地址可能相同吗?请详细说明原因

虚拟地址可能相同，但物理地址不可能相同。每一个进程有独立的虚拟地址空间,一个进 程中的虚拟地址需要经过转换才能转成物理地址。不同进程所对应的虚拟地址即使相同 也不会被转成统一物理地址。实际上内存是按照页来组织的，每进程有一个虚拟地址到 物理地址的映射表（页表），专门用于进程対应的虚拟页到物理页的映射（cpu内部有一 个TLB部件专门用于加速映射过程）。当进程A读写变量a B寸,会根据a的虚拟地址找到 其所在的虚拟页，通过页表找到物理页并进一步定位到物理地址。由于进程A、B拥有完 全不同的映射表，因此物理地址不会相同。答出虚拟地址和物理地址不同且知道虚拟地 址可以相同的给3分。

描述出映射过程的给4分。答出页表和页的给3分。

1. 有一个长度为n的整形数组,请给出判断某整数是否在该

数组中的方法。要求

1时间复杂度低于O（n）

2不要使用标准库中的提供的数据结构

3允许使用额外内存

4允许对原始数组进行预处理

5请在代码中添加必要注释。

使用set或unordered set红黑树或哈若

评分点

1正确对数组做有序化处理或树处理的给10分（如果用哈希表，要求对哈希函数进行实 现，但不对哈希函数的均匀性做要求，有效即可，但不得过于简单）

2正确实现查找部分逻辑的给10分（如折半查找或在树中查找）;如果用哈希表则需要哈 希碰撞处理

3代码风格糟糕可适当减分（最多可减5分）

1. 编写代码在一个给定的非空字符串后面追机可能少的字 符，使其成为个回串（正序与逆序完全相司），

如的追加a变为回串a,如给定字符串已经是回文串，则不需要 再追加。请在代码中添加要注释。

找到以最后一个字符串结尾的最长回文子串

评分点

1可以将原始字符串转化为非最短回文串的给5分（比如不经判断直接在后面迫加n-1 个字符的,n为给定字符串的长度）

2正确实现最短回文串的的给满分（如逐步确定包含未尾字符的最长回文子串,对复杂 度不做特别要求

3代码风格糟糕可适当减分（最多可减5分）

1. Session、Cookie、cache 的区别？

Session是单用户的会话状态。当用户访问网站时，产生一个sessionido并存在于 cookies中。每次向服务器请求时，发送这个cookies,再从服务器中检索是否有这个 sessionid保存的数据；

Cookie同session —样是保存你个人信息的，不过是保存在客户端，也就是你使用的 电脑上，并且不会被丢掉，除非你删除浏览器Cookie;

cache则是服务器端的緩存，是所有用户都可以访问和共享的，因为从Cache中读数据 比较快，所以有些系统（网站）会把一些经常被使用的数据放到Cache里，提高访问速 度，优化系统性能。

1. 在linux上，创建socket成功时会得到一个?

fd

1. 请简述TCP和UDP的区别？

KP是传输控制协议,提供的是面向连接、可靠的字节流服务。通信双方彼此交换数据 前,必须先通过三次握手协议建立连接，之后才能传输数据。TCP提供超时重传，丟弃重 复数据，检验数据，流量控制等功能，保证数据能从一端传到另端。UDP是用户数据报协 议,是一个简单的面向无连接的协议。UDP不提供可靠的服务。在数据数据前不用建立 连接故而传输速度很快。UDP主要用户'流環体传输,IP电话等对数据可靠性要求不是很 高的场合

1. 如果有几千个session,怎么提高效率？

把session放到redis或memcache等此类内存緩存中或着把session存储在SSD硬 盘上。

session对应的文件有一个特点就是小，一般在几KB左右，

如果session以文件方式存储，如果并发数量级有几千个，

此时系统硬盘的随机10早已成了系统中的最大瓶颈了，因为会话文件

是存储在多个小文件中，映射到存储空间不是一段连续的地址范围

所以硬盘的随机读职能力显得非常重要,而觉机械硬盘的随机10 一般在100/iops上下, (IOPS (Input/Output Operations Per Second),即每秒进行读写(I/O)操作的次数) SSD固态硬盘可以达几百至上千，所以在这么高并发读写的情况下如果无条件用SSD固 木舟

心、IHL

可以把session放到redis或memcache等内存緩存中，系统对内存的操作又是非常 快的，

只要你的内存足够大，再多session并发速度一样不会慢。

1. session是存储在什么地方，以什么形式存储的？

session变量保存在网叶服务器中，你不能直接修改。当然，调用程序中的 setAttributeO方法当然可以了。cookie存储的可不是具体的数据，要不岂不是太不 安全了，谁都可以修改session变量了,网站也臺无安全性可言。实际，在cookie中， 存储的是一个sessionld,它标示了一个服务器中的session变量，通过这种方式，服 务器就知道你到底是那个session 了。顺便说一句，如果客户端不支持cookie, session 也是可以实现的，在服务器端通过urlEncoder,可以实现sessionld的传递。所以， 记住客户端只存储session标识，实际内容在网页服务器中。

1. 索引的优点和缺点.

优点

1. 只大加快数据的检索速度；
2. 创建唯一性索引，保证数据库表中每一行数据的唯一性；
3. 加速表和表之间的连接；
4. 在使用分组和排序子句进行数据检索旺，可以显著减少查询中分组和排序的时间。 缺点
5. 索引需要占物理空间。
6. 当对表中的数据进行増加、删除和修改的时候，索引也要动态的维护，降低了数据的 维护速度。
7. 现在普通关系数据库用得数据结构是什么类型的数据结 构？

关系数据库采用的数据结构是：二维表

在关系型数据库系统中，所有的数据结构采用二维表的结构表示。

通常我们将这些二维表成为关系。在关系型数据库中，每一个关系都是一个二维表，无

论实体本身还是实体间的关系均用“关系”的二维表来表示。

而二维表就是我们现实世界中进行各种档案管理使用的方法，其中记录了大量的数据。

这样就用数学理论中的一个概念描述了现实世界的一个对象。

关系型数据库就是用关系描述数据的数据库系统。

（1）二维表与关系

关系可以用来描述二维表，对应的术语是：

关系--二维表；

元组二维表中的行

分量—二维表中的列

（2 ）二维表与关系型数据库中的数据

—个关系型数据库中的数据对应于一个二维表，其中对应的术语是：

二维表个数据库中的表、一个数据视图

二维表的行数据表中的记录

二维表的列--表记录的字段

1. 两个长度不超过80的字符串，串中包括大写、小写和空 格字符，去掉空格字符并忽略大小写后判断两个字符串是否 相等。

输入描述：第一行输入的是测试数据的组数n,每组测试数据占两行，第一行是第一个 字符申S1,第二行是第二个字符申S2o

输出描述：如果两个字符串相等，则输出YES,否则输出旳。

样例输入：

2

QW erTyu 0P

qwert YUOp

ASDDDfghj

asDDFgHJ

样例输出：

YES

NO

本題主要用到字符串处理函数str lens strlwzs strcmps strcpy

#include<st dio. h>

#include<st ring. h>

#define N 40

int main。

[

int i, j,乌 m=0:

char str[80], arr[N] [80], ch[N] [80]:

int len[80];〃用来存储每个字符串的长度

scanf 血):〃输入删试的组数 getchar () ;//皆除

for(i=0:i<2\*n:i++)

{ //输入n组字符串

gets (str);

strlwr(str) ;//把输入字符串中所有字符统一変为小写 len[m++] =strlen(str) ;//统计每个字符串的长度 strcpy(arr [i], str);〃把输入的字符串复制给arr数组

for(i=0;i<2\*n;i++)〃作用、着除每个字符串的空格 [

int k=0;

for(j=0; j<len[i] ; j++)

[

if(arr[i][j]!='')

{

ch[i] [k++]=arr[i] [j]:

ch[i] DdW ://标志字符串结尾

for(i=0;i<2\*n;i+=2)//比较每组字符串是否相等 !

if (st rcmp(ch[i], ch[i+l] )==0) {

printf("YES\n"):

}else{

printf("N3\n"):

return 0:

1. 归并排序的实现.

归并排序（MERGE-SORT）是利用归并的思想实现的排序方法，该算法采用经典的分治

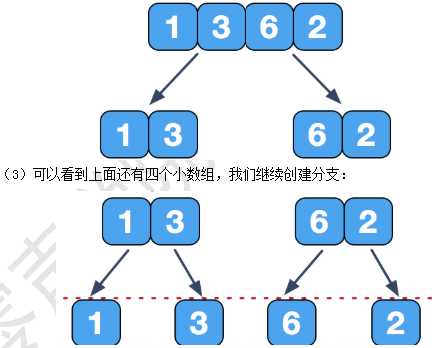
（divide-and-conquer ）策略（分治法将问题分（divide）成一些小的问题然后递归求解＞ 而治（conquer）的阶段则将分的阶段得到的各答案〃修补〃在一起，即分而治之）。 归并排序中，我们会先找到一个数组的中间下标mid,然后以这个mid为中心，对两边 分别进行排序，之后我们再根据两边已拝好序的子数组，重新进行值大小分配。 我们就以下面的数组为例：



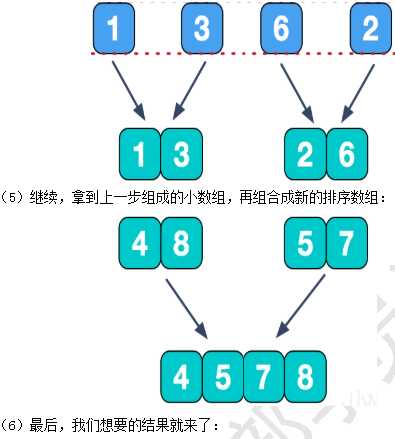
（1）mid等于数组最大下标/ 2 = 3,于是我们以下标3 （值为7）为中心，分别排序

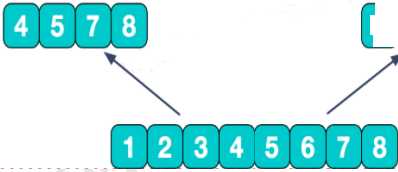
0~3和4曾的数字。

（2 ）此时两边的数组还可以创建分支:



（4）可以看到，此时我们已经无法继续创建分支了，因为数组数量为1时是没有必要 进行排序的，这时我们可以开始进行对冬一个小的数组进行排序并且合并：





代码如下：

public class MergeSort {

public static void mergeSort (int □ arr) { if (arr == null | | arr. length < 2) { return;

}

mergeSort (arr, 0, arr. length - 1):

}

public static void mergeSort (int □ arr, irct 1, int r) { if (1 == r) {

return;

int mid = 1 + ((r - 1) » 1):

mergeSort (arr, 1, mid):

mergeSort (arr, mid + 1, r):

merge (arr, 1, mid, r):

}

public static void merge(irrt [] arr, int 1, irrt m, int r) {

int [] help = new int [r - 1 + 1]：

int i = 0;

int 11 = 1;

int r1 = m + 1;

while (11 <= m && rl <= r) {

help[i++] = arr[11] < arr[rl] ? arr[11++] : arr[r1++]:

}

while (11 <= m) {

help[i++] = arr[11++]:

}

while (rl <= r) {

help [i++] = arr[r1++]:

}

for (int j = 0; j < help, length; j++) {

arr[l + j] = help[j]:



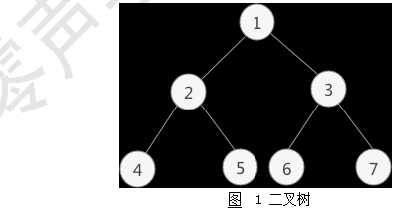
1. 二叉树先序遍历（递归与非递归）及C语言实现.

二叉树先序遍历的实现思想是：

访问根节点；

访问当前节点的左子树；

若当前节点无左子树，则访问当前节点的右子树;



以图1为例，采用先序遍历的思想遍历该二叉树的过程为: 访问该二叉树的根节点，找到1;

访问节点1的左子树，找到节点2;

访问节点2的左子树，找到节点4;

由于访问节点4左子树失败，且也没有右子树，因此以节点4为根节点的子树遍历完 成。但节点2还没有遍历其右子树，因此现在开始遍历，即访问节点5;

由于节点5无左右子树，因此节点5遍历完成，并且由此以节点2为根节点的子树 也遍历完成。现在回到节点1 ,并开始遍历该节点的右子树，即访问节点3;

访问节点3左子树，找到节点6;

由于节点6无左右子树，因此节点6遍历完成，回到节点3并遍历其右子树，找到 节点

节点7无左右子树，因此以节点3为根节点的子树遍历完成，同时回归节点1。由于 节点1的左右子树全部遍历完成，因此整个二叉树遍历完成；

因此，图1中二叉树采用先序遍历得到的序列为：1 2 4 5 3 6 7 逢归城：

二叉树的先序遍历采用的是逢归的思想，因此可以逢归实现。

#include <stdio. h>

#include 〈string. h>

/define TElemType int

//构渣结点的结构体

type def struct BiTNode {

TElemType data;//数据域

struct BiTNode \*lchild, \*rchild;//左右孩子指针

} BiTNode, \*BiTree;

〃初始化树的函数

void Creat eBilree (BiTree \*T) {

\*T= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T)->data=l;

(\*T) ->lchild= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T)->rchild= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T)->lchild->data=2;

(\*T) ->1 chi 1 d->Ichild= (BiTNodc\*) mal 1 oc (sizeof (BiTNode)):

(\*T) ->lchild->rchild= (BiTNode\*) malloc (sizeof (BiTNode)):

(\* T) - >1 chi 1 d- > rchil d-> dat a=5:

(\* T) - >1 chi 1 d- > rchil d-> 1 chi ld=NULL;

(\*T)->lchild->rchild->rchi ld=NULL;

(\*T)->rchild->data=3:

(\*T)->rchi 1 d->Ichild= (BiTNodc\*) mal 1 oc (sizeof (BiTNode)):

(\* T) - >r chi 1 d- > Ichil d-> dat a=6:

(\*T)->rchild->lchild->lchi ld=NULL;

(\*T)->rchild->lchild->rchi ld=NULL;

(\*T)->rchi 1 d->rchild= (BiTNodc\*) mal 1 oc (sizeof (BiTNode)):

(\* T) - >r chi 1 d- > rchil d-> dat a=7;

(\*T)->rchild->rchild->lchi ld=NULL;

(\*T)->rchild->rchild->rchi ld=NULL;

(\*T)->lchild->lchild->data=4 :

(\*T)->lchild->lchild->lchi ld=NULL;

(\*T)->lchild->lchild->rchi ld=NULL;

//模拟操作结点元素的函数，输出结点本身的数值

void displzyElemffiiTNode\* elem) {

printf ("%d ", elem->data):

}

〃先序遍历

void PreOrderTraverse (BiTree T) {

if (T) {

displ咨Elem(T) ;//调用操作结点数据的函数方法

PreOrderTraverse (T->lchild);//访问该结点的左铉子

PreOrderTraverse (T->rchild);//访问该结点的右莅子

}

〃如果结点为空，返回上一层

return;

}

int main () {

BiTree Tree:

CreateBiTreeUTree):

prmtfC先序遍历：")：

PreOrderTraverse (Tree):

}

运行结果：

先序遍历：1 2 4 5 3 6 7

非递归城：

而通归的底层实现依靠的是栈存儲结构，因此，二叉树的先序遍历既可以直接采用逢归 思想实现，也可以使用栈的存储结构模拟逢归的思想实现。

#include <stdio. h>

#include 〈string. h>

/define TElemType int

int top=-l://top変熨时刻表示栈顶元素所在位置

//构渣结点的结构体

type def struct BiTNode )

TElemType data;//数据域

struct BiTNode \*lchild, \*rchild;//左右孩子指针

} BiTNode, \*BiTree:

//初始化树的函数

void CreateBilree(BiTree \*T) {

\*T= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T)->data=l;

(\*T)->lchild= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T)->rchild= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T)->lchild->data=2:

(\*T)->1 chi 1 d->Ichild= (BiTNodc\*) mal 1 oc (sizeof (BiTNode)):

(\*T)->1 chi 1 d->rchild= (BiTNodc\*) mal 1 oc (sizeof (BiTNode)):

(\*T) - >lchild->rchild->dat a=5:

(\*T) - >lchild->rchild->lchi ld=NULL:

(\*T) - >lchild->rchild->rchi ld=NULL:

(\*T)->rchild->data=3:

(\*T) ->rchild->lchild= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T) ->rchild->lchild->dat a=6:

(\*T) ->rchild->lchild->lchi ld=NULL:

(\*T) ->rchild->lchild->rchi ld=NULL:

(\*T) ->rchild->rchild= (BiTNode\*) malloc (sizeof (BiTNode)):

(\*T) ->rchild->rchild->dat a=7:

(\*T) ->rchild->rchild->lchi ld=NULL:

(\*T) ->rchild->rchild->rchi ld=NULL;

(\*T) - >lchild->lchild->dat a=4;

(\*T) - >lchild->lchild->lchild=NULL:

(\*T) - >lchild->lchild->rchild=NULL:

}

//前序遍历使用的进栈函数

void push(BiTNode\*\* a, BiTNode\* elem) {

a[++top]=e]em：

}

〃弹栈函数

void pop( ) {

if (top=—l) {

return :

}

top--:

}

//模拟操作结点元麦的函数，输出结点本身的数值

void displz^Elem(BiTNode\* elem) {

prirrtf ", elem->data):

}

//拿到栈顶元素

BiTNode\* get Top (BiTNode\*\* a) (

return a [top]:

}

//先序遍历非递归算法

void PreOrder Traverse (BiTree Tree) {

BiTNode\* a[20]:〃定义一个顺序栈

BiTNode \* p;〃临时指针

push (a. Tree);〃根结点进栈

while (t op! =-1) {

p= get Top (a);〃取栈顶元素

pop();〃弹栈

while (p) {

零声学「謐理出品

更多、更全大厂面试姿料加Q群：762073882 displzyElem(p) ;//调隹结点的操作函数

〃如果该结点有右莅子，右孩子进栈

if (p->rchild) {

push (a, p->rchild):

}

p=^>lchild;//-直指向根结点最后一个左孩子

int main() {

BiTree Tree：

CreateBiTreeUTree): printff先序遍历：")： PreOrderTraverse (Tree): } 运行结果 先序遍历：1 2 4 5 3 6 7

1. 快速排序的思想、时间复杂度、实现以及优化方法？

，呻F序的三借麝

1. 选择基准：在待排序列中，按照某种方式挑出一个元素，作为’基准(pivot);
2. 分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准 左边的元素都比该基准小，在基准右边的元素都比基准大：
3. 通归地对两个序列进行快速排序，直到序列为空或者只有一个元素。

驟6^：

对于分治算法，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效 率会达到最大。

艮卩：同一数组，时间复杂度最小的是每次选取的基准都可以将序列分为两个等长的；时 间复杂度最大的是每次选择的基准都是当前序列的最大或最小元素；

橱妣唳见：

—般选择序列的第一个作为基数，快排代码如下：

void quicksort (vector<int> *&v,* int left, irrt right)

{

if (left < right)//false 递归结束

[

int key=v[left];〃基数賦值

int low = left:

int high = right:

while (low < high) //当low=high时，表示一抡分割结束

while (low < high &&〈[high] >= key)[low]为基数，从后向前 与基数比较

high—:

}

swzp (v[low], v [high]):

while (low < high && v[low] <= key)[high]为基数，从前向后 与基数比较

!

lowH-：

}

swzp (v[low], v [high]):

}

〃分割后，对每一分段重复上述操作

quicksort (v, left, low-1):

quicksort (v, lowH, right):

}

}

注：上述数组或序列v必须是引用类型的形参，因为后续快排结果需要直接反映在原序 列中；

优化：

上述快排的基数是序列的第一个元素，这样的对于有序序列，快排时间复杂度会达到最 差的o(n'2)o所以，优化方向就是合理的选择基数。

常见的做法“三数取中〃法(序列太短还要结合其他排序法，如插入排序、选择排序等), 如下：

1. 当序列区间长度小于7时，采用插入排序；
2. 当序列区间长度小于40时，将区间分成2段，得到左端点、右端点和中点，我们对 这三个点取中数作为基数；
3. 当序列区间大于等于40时，将区间分成8段，得到左三点、中三点和右三点，分 别再得到左三点中的中数、中三点中的中数和右三点中的中数，再将得到的三个中数取 中数，然后将该值作为基数。

具体代码只是在上一份的代码中将“基数赋值"改为①②③对应的代码即可：

int key=v[left];//基数賦值

if (right-left+1<=7) {

insertion\_sort (v, left, right) ://插入排序

return;

}else if (right-left+l<=8) {

key= SelectPivotOfThree (v, left, right);〃三个取中

}else(

〃三组三个取中，再三个取中(使用4次SelectPivotOfThree，此处不 具体展示)

1

需要调用的函数：

//插入排序算法

void insertion\_sort (vector<irrt> feunsorted, irrt left, irrt right) {

for (int i = left+1: i <= right; i++)if (unsorted[i - 1] > unsorted[i])

!

int temp = unsorted[i];

int j = i;

while (j > left && unsorted[j - 1] > temp) [

unsorted[j] = unsorted[j - 1]:

j--：

1

unsorted [j] = tenp:

〃三数取中，

同时将中值移到序列第一位

int SelectPivotOf Three (vector<int> &arr, int low, irrt high) { int mid = low + (high - low)/2;//计算数组中间的元素的下标 〃使用三数取中法迭择枢轴

if (arr[mid]

arr[high])//目标：arr[mid]

<=arr[high]

sw亜(arr [mid], arr [high]):

if

I

arr [low] > arr [high])//目标：arr [low]

<=

arr [high]

sw邳(arr [low], arr [high]):

if (arr [mid] > arr[low]) //目标：arr[low]

>=arr[mid]

sw亜(arr [mid], arr [low]):

〃此时，arr[mid] <= arr[low] <= arr[high]

return arr [low]:

//low的位置上保存这三个位置中间的值

//分割时可以直接使用low位置的元素作为枢轴，而不用改变分割函数了

}

这里需要注意的有两点： ①插入排序算法实现代码;

②三数取中函数不仅仅要实现取中，还要将中值移到最低位，从而保证原分割函数依然 可用。

1. 将“引用”作为函数参数有哪些特点？
2. 传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调 函数中的实参变里或对象的一个别名来使用,所以在被调函数中对形参变里的操作就是 对相应的目标对象(在主调函数中)的操作。
3. 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作量 传递函数当发生函数调用时，需要给形参分配存储单元是实参变里的副本；如果传递的 是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量 传递参数的效率和所占空间都好。(3)使用指针作为函数的参数虽然也能达到与使用引 用的效果，但是，在被调函数中同样要给形参分配存储单元,且需要重复使用指针变里名 的形式进行运算,这很容易产生错误且程序的阅读性较差;另一方面，在主调函数的调用 点处，必须用变里的地址作为实参。而引用更容易使用，更清晰。
4. 顺序栈的表示和实现

顺序栈*，*即栈的顺序存储结构是利用一组池址连续的存储单元依次存放自栈底到栈顶的 数据元素,同时附设指针top指示栈顶元素在顺序栈中的位置。通常的习惯做法以top=0 表示空栈。一般来说，在初始化设空栈旺不应限定栈的最大容量。一个较合理的做法: 先为栈分配一个基本容量，然后在应用近程中，当栈的空间不足在进行扩展。

#define ST^K\_INIT\_SIZE 100

#define ST骯KINCREMENT 10

type def struct SqStack

[

int \*base;

int \*top:

int stacksize;

} SqStack;

int Init St ack (SqStack &S)

{

S. base=(int \*)malloc (ST^CK\_INIT\_SIZE\*sizeof(int)):

if (IS.base)

exit (0):

*S.* top=S. base;

5. st acksize=ilJ¥JK\_lNH\_51ZE:

return 0:

1

int Get Top (SqStack S, int &e)

[

if (S. top=S.base)

return 1:

e=\*(S.top-l):

return 0:

int Push (SqStack *&S,* int e)

[

if (S. top-S. base>=S. stacksize)

[

S. base= (int\*) realloc (S. base, (S. st acks ize+ST ACK INCREMENT) \* sizeof(irrt));

if (! S.base)

return 1:

S. top=S. base+S. stacksize;

S. st acksize-fcSTACKINCREIOT:

}

\*S.top=e:

S. top++; return 0;

}

int Pop (SqStack &S, irrt &e)

[

if (S. top=S.base)

return 1;

—Stop;

e=\*S. top:

return 0;

}

int StackEmpty (SqSt ack S)

[

if (S. top=S.base)

return 1:

return 0:

}

int Dest roySt ack (SqStack &S)

{

free (S. base):

S.top=NULL:

S. base=S.top;

return 0:

}

1. 请描述索引的特征？

索引有两个特征，即唯一性索引和复合索引。

唯一性索引保证在索引列中的全部数据是唯一的，不会包含冗余数据。如果表中已经 有一个主键约束或者唯一性键约束，那么当创建表或者修改表时，SQL Server自动创 建一个唯一性索引。然而，如果必须保证唯一性，那么应该创建主键约束或者唯一性键 约束，而不是创建一个唯一性索引。当创建唯一性索引时，应该认真考虑这些规则： 当在表中创建主键约束或者唯一性键约束时，SQL Server g动创建一个唯一性索引； 如果表中已经包含有数据，那么当创建索引时，SQL Server检查表中已有数据的冗余 性；每当使用插入语句插入数据或者使用修改语句修改数据时，SQL Server检查数据 的冗余性：如果有冗余值，那么SQL Server 消该语句的执行，并且返回一个错误消 息;确保表中的每一行数据都有一个唯一值,这样可以确保每一个实体都可以唯一确认; 只能在可以保证实体完整性的列上创建唯一性索引，例如，不能在人事表中的姓名列 上创建唯一性索引，因为人们可以有相同的姓名。

复合索引就是一个索引创建在两个列或者多个列上。在搜索时，当两个或者多个列作 为一个关键值时，最好在这些列上创建复合索引。当创建复合索引时，应该考虑这些规 则：最多可以把16个列合并成一个单独的复合索引，构成复合索引的列的总长度不能 起过9。。字节，也就是说复合列的长度不能太长；在复合索引中，所有的列必须来自同 —个表中，不能跨表建立复合列；在复合索引中，列的排列顺序是非常重要的，因此 要认真排列列的顺序，原则上，应该首先定义最唯一的列，例如在(C0L1, C0L2)上的 索引与在(C0L2, C0L1)上的索引是不相同的，因为两个索引的列的顺序不同；为了 使查询优化器使用复合索引，查询语句中的雌职子句必须参考复合索引中第一个列； 当表中有多个关键列时，复合索引是非常有用的；使用复合索引可以提高查询性能，减 少在一个表中所创建的索引数量。

1. 如何理解聚簇索引和非聚簇索引的的体系结构？

聚篠索引的体系结构

索引的结构类似于树状结构，树的顶部将为叶级，树的其它部分称为非叶级，树的根部 在非叶级中。同样，在聚簇索引中，聚簸索引的叶级和非叶级构成了一个树状结构， 索引的最低级是叶级。在聚簇索引中，表中的数据所在的数据页是叶级，在叶级之上的 索引页是非叶级，索引数据所在的索引页是非叶级。在聚簇索引中，数据值的顺序总 是按照升序排列。

应该在表中经常搜索的列或者按照顺序访问的列上创建聚簇索引。当创建聚簇索引时, 应该考虑这些因素：每一个表只能有一个聚簇索引，因为表中数据的物理顺序只能有 —个；表中行的物理顺序和索引中行的物里顺序是相同的，在创建任何非聚簇索引之前 创建聚簇索引，这是因为聚簇索引改变了表中行的物理顺序，数据行按照一定的顺序 排列，并且自动维护这个顺序；关键值的唯一性要么使用UNIQUE关键字明确维护，要 么由一个内部的唯一标识符明确维护，这些唯一性标识符是系统自己使用的，用户不 能访问；聚簇索引的平均大小大约是数据表的百分之五，但是，实际的聚簇索引的大小 常常根据索引列的大小变化而变化；在索引的创建过程中，SQL Server 1^时使用当前 数据库的磁盘空间，当创建聚簇索引时，需要1.2倍的表空间的大小，因此，一定要保 证有足够的空间来创建聚簇索引。

当系统访问表中的数据时，首先确定在相应的列上是否存在有索引和该索引是否对要 检索的数据有意义。如果索引存在并且该索引非常有意义，那么系统使用该索引访问 表中的记录。系统从索引开始浏览到数据，索引浏览则从树状索引的根部开始。从根部 幵始，搜索值与每一个关键值相比较，确定搜索值是否大于或者等于关键值。这一步 重复进行，直到碰上一个比搜索值大的关键值，或者该搜索值大于或者等于索引页上所 有的关键值为止。

非聚篠索引的体系结构

非聚簇索引的结构也是树状结构，与聚簸索引的结构非常类似，但是也有明显的不同。 在非聚簇索引中，叶级仅包含关键值，而没有包含数据行。非聚簇索引表示行的逻辑顺 序。非聚簇索引有两种体系结构：一种体系结构是在没有聚簇索引的表上创建非聚簇 索引，另一种体系结构是在有聚簇索引的表上创建非聚簇索引。

如果一个数据表中没有聚簇索引，那么这个数据表也称为数据堆。当非聚簇索引在数 据堆的顶部创建时，系统使用索引页中的行标识符指向数据页中的记录。行标识符存 储了数据所在位置的信息。数据堆是通迂使用索引分配图＜M）页来维护的。L娜页 包含了数据堆所在簇的存储信息。在系统表sysindexes中，有一个指针指向了与数据 堆相关的第一个L娜页。系统使用L娜页在数据堆中浏览和寻找可以插入新的记录行的 空间。这些数据页和在这些数据页中的记录没有任何的顺序并且也没有链接在一起。 在这些数据页之间的唯一的连接是L娜中记录的顺序。当在数据堆上创建了非聚簇索引 时，叶级中包含了指向数据页的行标识符。行标识符指定记录行的逻辑顺序，由文件 ID、页号和行ID组成。这些行的标识符维持唯一性。非聚簇索引的叶级页的顺序不同 于表中数据的物理顺序。这些关键值在叶级中以升序维持。

当非聚簇索引创建在有聚簇索引的表上的时候,系统使用索引页中的指向聚簇索引的聚 簇键。聚簇键存储了数据的位置信息。如果某一个表有聚簇索引，那么非聚簇索引的 叶级包含了映射到聚簇键的聚簇键值，而不是映射到物理的行标识符。当系统访问有非 聚簇索引的表中数据时，并且这种非聚熊索引创建在聚簇索引上，那么它首先从非聚 簇索引来找到指向聚簇索引的指针，然后通过使用聚簇索引来找到数据。

当需要以多种方式检索数据时，非聚簇素引是非常有用的。当创建非聚簇索引时，要考 虑这些情况：在缺省情况下，所创建的索引是非聚簇索引；在每一个表上面，可以创建 不多于249个非聚簇索引,而聚簇索引最多只能有一个。

系统如何访问表中的数据

—般地，系统访问数据库中的数据，可以使用两种方法：表扫描和索引查找。第一种 方法是表扫描，就是指系统将指针放置在该表的表头数据所在的数据页上，然后按照 数据页的排列顺序，一页一页地从前向后扫描该表数据所占有的全部数据页，直至扫描 完表中的全部记录。在扫描时，如果找到符合查询条件的记录，那么就将这条记录挑 选出来。最后，将全部挑选出来符合查询语句条件的记录显示出来。第二种方法是使用 索引查找。索引是一种树状结构，其中有储了关键字和指向包含关键字所在记录的数 据页的指针。当使用索引查找时，系统沿着索引的树状结构，根据索引中关键字和指针, 找到符合查询条件的的记录。最后，将全部查找到的符合查询语句条件的记录显示出 来。

在SQL Server中，当访问数据库中的数据时，由SQL Server确定该表中是否有索引存 在。如果没有索引，那么SQL Server使用表扫描的方法访问数据库中的数据。查询处 理器根据分布的统计信息生成该查询语句的优化执行规划,以提高访问数据的效率为目 标，确定是使用表扫描还是使用索引。

1. C和C++有什么不同?

从机制上:C是面向过程的（但C也可以编与面向对象的程序）。c+4是面向对象的，提供 了类。

但是,c卄编写面向对象的程序比c容易。

从适用的方向:c适合要求代码体积小的,效率高的场合，如嵌入式;C-适合更上层的复 杂的；linux核心

大部分是c写的，因为它是系统软件，效率要求极高从名称上也可以看出,c卄比c 多了 +说明

crH是c的超集;那为什么不叫c+而叫c+磯,是因为c卄比C来说扩充的东西太多 了,所以就

在c后面放上两个+；于是就成了 C++。

C语言是结构化编程语言,CH是面向对象编程语言。LUPA开源社区｝ n\*r2C/J.!8f

Ci侧重于对象而不是过程,侧重于类的设计而不是逻辑设计.

1. 头文件的作用时什么？

一、 通过头文件来调用库功能。在很多场含，源代码不便(或不准)向用户公布，只要向用 户提供头文件和二进制的摩即可。用户只需要按照头文件中的接口声明来调用库功能而 不必关心接口怎么实现的。编译器中提取相应的代码。

二、 头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中 的声译器就会指出错误简单的规则能大大碱轻程序员调试、改错的负担。

1. memcpy函数实现.

memcpy函数作用：

将由灯c指向地址为起始地址的连续n个字节的数据复制到以de"指向地址为起始地 址的空间内，函数返回一个指向dest的指针

舫嫩：

1- src和de st所指内存区域不能重囊

1. 与strcpy相比，memcpy遇到并不会结束，而是一定会拷贝完n个字节
2. memcpy可以拷贝任何数据类型的对象,可以指定拷贝的数据长度
3. 如果dest本身就有数据，执行memcpy ()后会覆盖原有的数据
4. de"和src都不一定时数组，任意的可读写的空间均可
5. 如果要追加数据*，*则每次执行memcpy后，要将目标数组地址増加到所要追加数据的 地址

弟见代码：

#define \_CRT\_SECURE\_NO\_WARNINSS 1

#include<st dio.h>

#include<stdlib. h>

void \* myjnemcpy (void \* de st, void \*src, unsigned count )

[

if (dest == NULL || src == NULL)

return NULL： char\* pdest = (char\*) dest:

char\* psrc = (char\*) src:

while (count--)

I

\*pdest++ = \*psrc++；

}

return dest:

}

int main。

[

char src[] = "hello":

char dest [] = world":

my\_memcpy(dest, src, st rlen(src)): prirrtf("%s\ dest);

syst em (“ pause "):

return 0:

1. 寻找二叉树中两个节点的最近公共祖先节点

—、该二叉树为搜索二叉树

搜索二叉树的特点：

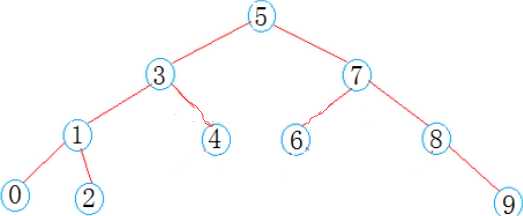
任意一个节点的左子树的所有节点值都比该节点的值小，其右子树的所有节点值都比该 节点的值大。

解决该问题方法：

从树的根节点开始和两个节点作比较，殳戚当前节点的值比两个节点的值都大，则这两 个节点的最近公共祖先节点一定在该节点的左子树中，则下一步遍历当前节点的左子 树；

如果当前节点的值比两个节点的值都小，则这两个节点的最近公共祖先节点一定在该节 点的右子树中，下一步遍历当前节点的右子树；这样直到找到第一个值是两个输入节点 之间的值的节点，该节点就是两个节点的最近公共祖先节点。

如图：



二、该二叉树为一般二叉树，有二叉树节点中包含指向父节点的指针

struct BinaryNode

[

Binarj^Iode\* \_left:

Binarj^Iode\* \_right;

Binarj^Iode\* \_parerct:

int \_value:

Binarj^Iode (const int& value)

:\_value (value)

,\_left (NULL)

,.right (NULL)

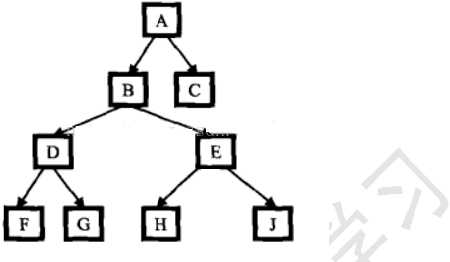
,\_parerrt (NULL)

{}

}：

首先给出nodel的父节点nodel-^arent,然后将nodel的所有父节点依次和 node2->pazent作比较，如果发现两个节点相等，则该节点就是最近公共祖先，直接将 其返回。如果没找到相等节点，则将node2的所有父节点依次和 nodel~>\_parent->\_paxent 作比较 直到 nodel-

>\_parent==NULLo



代码如下：

Binarj^Iode \* GetLastCommonAncestor (BinaryNode \* root, Binarj^Iode \* node 1,

Binarj^Iode \* node2)

[

Binarj^Iode \* temp:

while (node 1 != NULL)

[

nodel = nodel->\_parent;

t enp = node 2;

while (temp != NULL)



if (node 1 == t emp->\_parent)

return node 1:

tenp = temp->\_parent:





1. 死锁产生的条件，以及如何避免死锁，银行家算法，产

生死锁后如何解决？

产生死锁的四个必要条件（互请不循）：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件：若干进程之间形成一种头尾相接的【循环等待资源】关系。 避免死锁：

（1） .按同一顺序访问对象。（注：避免出现循环）

（2） .避免事务中的用户交互。（注：减少持有资源的时间，较少锁竟争）

（3） .保持事务简短并处于一个批处理中。（注：同（2）,减少持有资源的时间）

（4） .使用较低的隔离级别。（注：使用较低的隔离级别（例如已提交读）比使用较高的 隔离级别（例如可序列化）持有共享锁的时间更短，减少锁竟争）

（5） .使用基于行版本控制的隔离级别：

银行家算法

银行家算法是一个避免死锁的著名算法，它是以银行借贷系统的分配策略为基础，判断 并保证系统的安全运行。

总之:

个进程申请使用资源的时候，银行家算法通过【先试探分配给该进程资源】*，*然 后【通过安全性算法判断分配后的系统是否处于安全状态L若不安全则试探分配作废, 让该进程继续等待。

当一进程提出资源申请时，银行家算法执行下列步骤以决定是否向其分配资源：

1） 检查该进程所需要的资源是否已超过它所宣布的最大值。

2） 检查系统当前是否有足够资源满足该进程的请求。

3） 系统试探着将资源分配给该进程，得到一个新状态。

4） 执行安全性算法，若该新状态是安全的，则分配完成；若新状态是不安全的，则恢 复原状态，阻塞该进程。

假设资源P1申请资源，银行家算法先试探的分配给它（当然先要看看当前资源池中的 资源数量够不够）*，*【若申请的资源数量小于等于Available,然后接着判断分配给Pl 后剩余的资源，能不能使进程队列的某个进程执行完毕】*，*【若没有进程可执行完毕, 则系统处于不安全状态】（即此时没有f进程能够完成并释放资源，随时间推移，系 统终将处于死锁状态）。

若有进程可执行完毕，则假设回收已分配给它的资源（剩余资源数量増加）*，*把这个进 程标记为可完成，并继续判断队列中的其它进程，若所有进程都可执行完毕，则系统处 于安全状态，并根据可完成进程的分配顺序生成安全序列（如｛P0, P3, P2, P1｝表示将 申请后的剩余资源Work先分配给P0 -〉回收（Work+已分配给P0的A0=Work）- ＞分配 给P3 - ＞回收（Work+A3=Work）- ＞分配給P2 - ＞ 满足所有进程）。

60.在KMP算法中，已知模式串为ADABCADADA,请写出模 式串的next数组值.

'前缀'指除了最后一个字符以外，一个字符串的全部头部组合;'后缀'指除了第一个字 符以外，一个字符串的全部尾部组合。next数组值就是'前缀'和'后缀'的最长的共有元 素的长度。

首先求最大相同前缀后缀长度。*W*的前缀和后缀都为空集,没有共有元素，共有元素 长度为O;AD '的 前缀为[A],后缀为[D],没有共有元素，共芍元素长度为0/ ADA '的前缀为[A,.W],后缀 为[DA, A],共有元素为A,长度为1;' ADAB '的前缀为[出.W, ADA],后缀为[DAB, AB, B], 共有元素长度为。;以此类推，最大公共元素长度为：

0,1,0,0,1, 23,2。

然后将最长相同前缀后缀长度值右移一位,并将next [0]初值赋为-1,得到的next数组: -1,0,0,1,0,0,1, 2,3,2。

在某些语言中，数组不是从。开始索引的，而是从1开始索引，只需要将next数组中每个 值加L得到

0,1,1, 2,1,1, 2, 3,43。

所以结果为 0,1,1, 2,1,1, 2, 3,43

©ES I

202頌阿里精选 202何百度精选 202用腾汛精选

面涙题.pdf 题.pdf 面涙题.pdf

Q HK»a.mp4

Q TCPIP^HMfi ,—次ilff臼海向増ZTJ.EP4 ± &

D 么3W内存池.mp4

D 手锥8itB.mp4

Q react。,街HOSSi阪現關2iKS.mp4

Q nginxi狷一緩Hit啲实3i.mp4

Q MySQLS^S^fl作.mp4

Q i5并发 tcpip 网《8io.mp4

Q 去中国,p2p ,网塔乔『^8定mp4

Q 财Kt!能优化一^mS£mp4

Q 区蜒的IKS ,去机冷网响Sit.mp4

Q深入浅出UD\*原理及瞄分片方法.mp4

D线側朝mp4

D后台齢選程挂了毎么办.mp4