2022 年美团精选 50 面试题及答案

1. 在 ACM 竞赛中，一支队伍由三名队员组成，现在有 N+M 名学生，其中有 N 名学生擅长算法，剩下 M 名学生擅长编程，这些学生要参加 ACM 竞赛，他们的教练要求每支队伍至少有一名擅长算法和一名擅长编程的学生，那么这些学生最多可以组成多少支队伍？

输入： 输入两个整数 M，N，其中 1<N,M<10000000
输出： 最多可以组成的队伍数

```
#include <iostream>
using namespace std;

int main()
{
    int cnt = 0,n,m;
    cout << "输入 N 个擅长算法的，M 个擅长编程的：" << endl;
    cin >> n >> m;
    while(n!=0&&m!=0&&m+n!=2){
        if(n>=m){
            n = n-2;
            m = m-1;
            cnt++;
        }
        else if(n<m){
            m = m-2;
            n = n-1;
            cnt++;
        }
    }
    cout << "最大组对数量" << cnt << endl;
    return 0;
}
```

2. 什么是幂等性

幂等性概念：幂等通俗来说是指不管进行多少次重复操作，都是实现相同的结果。

3. REST 请求中哪些是幂等操作

GET，PUT，DELETE 都是幂等操作，而 POST 不是
分析
首先 GET 请求很好理解，对资源做查询多次，此实现的结果都是一样的。
PUT 请求的幂等性可以这样理解，将 A 修改为 B，它第一次请求值变为了 B，再进行多次此操作，最终的结果还是 B，与一次执行的结果是一样的，所以 PUT 是幂等操作。
同理可以理解 DELETE 操作，第一次将资源删除后，后面多次进行此删除请求，最终结果是一样的，将资源删除掉了。
POST 不是幂等操作，因为一次请求添加一份新资源，二次请求则添加了两份新资源，多次请求会产生不同的结果，因此 POST 不是幂等操作。

4. 根据幂等性区分 POST 与 PUT 的使用

可根据 idempotent（幂等性）做区分。
举一个简单的例子，假如有一个博客系统提供一个 Web API，模式是这样
http://superblogging/blogs/{blog-name}，很简单，将{blog-name}替换为我们的 blog 名字，往这个 URL 发送一个 HTTP PUT 或者 POST 请求，HTTP 的 body 部分就是博文，这是一个很简单的 REST API 例子。
我们应该用 PUT 方法还是 POST 方法？
取决于这个 REST 服务的行为是否是 idempotent 的，假如我们发送两个 http://superblogging/blogs/post/Sample 请求，服务器端是什么样的行为？如果产生了两个博客帖子，那就说明这个服务不是 idempotent 的，因为多次使用产生了副作用了嘛；如果后一个请求把第一个请求覆盖掉了，那这个服务就是 idempotent 的。前一种情况，应该使用 POST 方法，后一种情况，应该使用 PUT 方法。

5. CAS 的缺点及解决.

CAS 的缺点有如 ABA 问题，自旋锁消耗问题、多变量共享一致性问题.
1.ABA：
问题描述：线程 t1 将它的值从 A 变为 B，再从 B 变为 A。同时有线程 t2 要将值从 A 变为 C。但 CAS 检查的时候会发现没有改变，但是实质上它已经发生了改变 。可能会造成数据的缺失。
解决方法：CAS 还是类似于乐观锁，同数据乐观锁的方式给它加一个版本号或者时间戳，如 AtomicStampedReference
2.自旋消耗资源：
问题描述：多个线程争夺同一个资源时，如果自旋一直不成功，将会一直占用 CPU。

解决方法：破坏掉 for 死循环，当超过一定时间或者一定次数时，return 退出。JDK8 新增的 LongAddr,和 ConcurrentHashMap 类似的方法。当多个线程竞争时，将粒度变小，将一个变量拆分为多个变量，达到多个线程访问多个资源的效果，最后再调用 sum 把它合起来。

虽然 base 和 cells 都是 volatile 修饰的，但感觉这个 sum 操作没有加锁，可能 sum 的结果不是那么精确。

2.多变量共享一致性问题：

解决方法： CAS 操作是针对一个变量的，如果对多个变量操作，

1) 可以加锁来解决。

2) 封装成对象类解决。

6. B+树特点

(1)每个结点的关键字个数与孩子个数相等，所有非最下层的内层结点的关键字是对应子树上的最大关键字，最下层内部结点包含了全部关键字。

(2)除根结点以外，每个内部结点有 到 m 个孩子。[3]

(3)所有叶结点在树结构的同一层,并且不含任何信息(可看成是外部结点或查找失败的结点)，因此，树结构总是树高平衡的。

7. 详细解释事务的隔离性.

事务的隔离性

为了保证事务的隔离性，自然我们可以把事务设计成单线程的，这样的话效率就会极其低下，为了保证隔离性，又不失效率我们把丧失隔离性的情况分为三种。

脏读：读到另一个未提交事务的数据

幻读：在一个事务过程中已经读取了一次表，此时恰巧另一个事务 commit，导致这次事务再一次读取表时前后不一致。（表影响）

不可重复读：在一个事务过程中已经读取了一次 a 数据，此时恰巧另一个事务 commit，导致这次事务再一次读取 a 数据时前后不一致。

针对这三种情况推出了四大隔离级别

四大隔离级别:

Read uncommitted -- 不防止任何隔离性问题,具有脏读/不可重复度/虚读(幻读)问题

Read committed -- 可以防止脏读问题,但是不能防止不可重复度/虚读(幻读)问题

Repeatable read -- 可以防止脏读/不可重复读问题,但是不能防止虚读(幻读)问题

Serializable -- 数据库被设计为单线程数据库,可以防止上述所有问题

这四大隔离级别，安全性递增。效率递减

8. 已知一个函数 rand7()能够生成 1-7 的随机数，请给出一个函数，该函数能够生成 1-10 的随机数。

该解法基于一种叫做拒绝采样的方法。主要思想是只要产生一个目标范围内的随机数，则直接返回。如果产生的随机数不在目标范围内，则丢弃该值，重新取样。由于目标范围内的数字被选中的概率相等，这样一个均匀的分布生成了。

显然 rand7 至少需要执行 2 次，否则产生不了 1-10 的数字。通过运行 rand7 两次，可以生成 1-49 的整数，

```
   1 2 3 4 5 6 7
1  1 2 3 4 5 6 7
2  8 9 10 1 2 3 4
3  5 6 7 8 9 10 1
4  2 3 4 5 6 7 8
5  9 10 1 2 3 4 5
6  6 7 8 9 10 * *
7  * * * * * * *
```

由于 49 不是 10 的倍数，所以我们需要丢弃一些值，我们想要的数字范围为 1-40，不在此范围则丢弃并重新取样。

代码：

```
int rand10() {
  int row, col, idx;
  do {
    row = rand7();
    col = rand7();
    idx = col + (row-1)*7;
  } while (idx > 40);
  return 1 + (idx-1)%10;
}
```

由于 row 范围为 1-7，col 范围为 1-7，这样 idx 值范围为 1-49。大于 40 的值被丢弃，这样剩下 1-40 范围内的数字，通过取模返回。下面计算一下得到一个满足 1-40 范围的数需要进行取样的次数的期望值：

$$
\begin{aligned}
E(\#\ \text{calls to rand7}) &= 2 * (40/49) + \\
&\quad 4 * (9/49) * (40/49) + \\
&\quad 6 * (9/49)^2 * (40/49) + \\
&\quad \cdots \\
&= \sum_{k=1}^{\infty} 2k * (9/49)^{k-1} * (40/49) \\
&= (80/49) / (1 - 9/49)^2 \\
&= 2.45
\end{aligned}
$$

9. 请说明线程池都有哪些优化措施.

线程等待时间所占比例越高，需要越多线程。线程 CPU 时间所占比例越高，需要越少线程。
如果你是 CPU 密集型运算，那么线程数量和 CPU 核心数相同就好，避免了大量无用的切换线程上下文.
如果你是 IO 密集型的话，需要大量等待，那么线程数可以设置的多一些，比如 CPU 核心乘以 2.

10. C++11 创建线程的三种方式

1. 通过函数
thread：标准库的类
join：阻塞主线程并等待

```
// MultiThread.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include<iostream>
#include<vector>
#include<map>
#include<string>
#include<thread>

using namespace std;
void myPrint()
{
    cout << "线程开始运行" << endl;
    cout << "线程运行结束了" << endl;
}

int main()
{
    std::thread my2Obj(myPrint);  // 可调用对象
    my2Obj.join();// 主线程阻塞在这，并等待 myPrint()执行完
    cout << "wangtao" << endl;
    return 0;
}
```

detach(): 将主线程和子线程完全分离，子线程会驻留在后台运行,被 C++运行时库接管，失去控制

```
void myPrint()
{
```

```
    cout << "线程开始运行 1" << endl;
    cout << "线程开始运行 2" << endl;
    cout << "线程开始运行 3" << endl;
    cout << "线程开始运行 4" << endl;
    cout << "线程开始运行 5" << endl;
    cout << "线程开始运行 6" << endl;
    cout << "线程开始运行 7" << endl;
    cout << "线程开始运行 8" << endl;
    cout << "线程开始运行 9" << endl;

}

int main()
{
    std::thread my20bj(myPrint); // 主线程阻塞在这，并等待 myPrint()执行完
    my20bj.detach();
    cout << "wangtao1" << endl;
    cout << "wangtao2" << endl;
    cout << "wangtao3" << endl;
    cout << "wangtao4" << endl;
    cout << "wangtao5" << endl;
    cout << "wangtao6" << endl;
    cout << "wangtao7" << endl;
    cout << "wangtao8" << endl;
    return 0;
}
```

joinable()：判断是否可以成功使用 join()或者 detach()

程序说明：detach 后不能在实施 join

```
int main()
{
    std::thread my20bj(myPrint); // 主线程阻塞在这，并等待 myPrint()执行完
    if (my20bj.joinable()){
        cout << "1:joinable() == true" << endl;
    }
    else {
        cout << "1:joinable() == false" << endl;
    }
    my20bj.detach();

    if (my20bj.joinable()) {
        cout << "2:joinable() == true" << endl;
```

```
    }
    else {
        cout << "2:joinable() == false" << endl;
    }
    cout << "wangtao1" << endl;
    cout << "wangtao2" << endl;
    cout << "wangtao3" << endl;
    cout << "wangtao4" << endl;
    cout << "wangtao5" << endl;
    cout << "wangtao6" << endl;
    cout << "wangtao7" << endl;
    cout << "wangtao8" << endl;
    return 0;
}

int main()
{
    std::thread my20bj(myPrint); // 主线程阻塞在这，并等待 myPrint()执行完
    if (my20bj.joinable()){
        my20bj.join();
    }
    cout << "wangtao1" << endl;
    cout << "wangtao2" << endl;
    cout << "wangtao3" << endl;
    cout << "wangtao4" << endl;
    cout << "wangtao5" << endl;
    cout << "wangtao6" << endl;
    cout << "wangtao7" << endl;
    cout << "wangtao8" << endl;
    return 0;
}
```

2.通过类对象创建线程

```
class CObject
{
public:
    void operator ()() {
        cout << "线程开始运行" << endl;
        cout << "线程结束运行" << endl;
    }
};
```

```
int main()
{
    CObject obj;
    std::thread my2Obj(obj); // 主线程阻塞在这，并等待 myPrint()执行完
    if (my2Obj.joinable()){
        my2Obj.join();
    }
    cout << "see you " << endl;

    return 0;
}



class CObject
{
    int& m_obj;
public:
    CObject(int& i) :m_obj(i) {}
    void operator ()() { // 不带参数
        cout << "线程开始运行 1" << endl;
        cout << "线程开始运行 2" << endl;
        cout << "线程开始运行 3" << endl;
        cout << "线程开始运行 4" << endl;
        cout << "线程开始运行 5" << endl;
    }
};
int main()
{
    int i = 6;
    CObject obj(i);
    std::thread my2Obj(obj); // 主线程阻塞在这，并等待 myPrint()执行完
    if (my2Obj.joinable()){
        my2Obj.detach();
    }
    cout << "see you " << endl;

    return 0;
}
```

用 detach() 主线程结束对象即被销毁，那么子线程的成员函数还能调用吗？
这里的的对象会被复制到子线程中，当主线程结束，复制的子线程对象并不会被销毁
只要是没有引用、指针就不会出现问题

通过复制构造函数和析构函数来验证对象是否复制到了子线程中

```
// MultiThread.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include<iostream>
#include<vector>
#include<map>
#include<string>
#include<thread>
using namespace std;
class CObject
{
    int& m_obj;
public:
    CObject(int& i) :m_obj(i) {
        cout << "ctor" << endl;
    }
    CObject(const CObject& m) :m_obj(m.m_obj) {
        cout << "copy ctor" << endl;
    }
    ~CObject(){
        cout << "dtor" << endl;
    }
    void operator ()() { // 不带参数
        cout << "线程开始运行 1" << endl;
        cout << "线程开始运行 2" << endl;
        cout << "线程开始运行 3" << endl;
        cout << "线程开始运行 4" << endl;
        cout << "线程开始运行 5" << endl;
    }
};
int main()
{
    int i = 6;
    CObject obj(i);
    std::thread my2Obj(obj); // 主线程阻塞在这，并等待 myPrint()执行完
    if (my2Obj.joinable()){
        my2Obj.detach();
    }
    cout << "see you " << endl;

    return 0;
```

```
}
```

子线程的析构函数在后台执行，所以输出的 dtor 是主线程的。用 join() 结果为：

3.通过 lambda 表达式创建线程

```
int main()
{
    auto myLamThread = [] {
        cout << "线程开始运行" << endl;
        cout << "线程结束运行" << endl;
    };
    thread cthread(myLamThread);
    cthread.join();
    std::cout << "see you " << endl;

    return 0;
}
```

11. 什么是并行计算?

并行计算（Parallel Computing）是指同时使用多种计算资源解决计算问题的过程，是提高计算机系统计算速度和处理能力的一种有效手段。它的基本思想是用多个处理器来协同求解同一问题，即将被求解的问题分解成若干个部分，各部分均由一个独立的处理机来并行计算。并行计算系统既可以是专门设计的、含有多个处理器的超级计算机，也可以是以某种方式互连的若干台的独立计算机构成的集群。通过并行计算集群完成数据的处理，再将处理的结果返回给用户。

并行计算可分为时间上的并行和空间上的并行。

时间上的并行：是指流水线技术，比如说工厂生产食品的时候步骤分为：

1. 清洗：将食品冲洗干净。
2. 消毒：将食品进行消毒处理。
3. 切割：将食品切成小块。
4. 包装：将食品装入包装袋。

如果不采用流水线，一个食品完成上述四个步骤后，下一个食品才进行处理，耗时且影响效率。但是采用流水线技术，就可以同时处理四个食品。这就是并行算法中的时间并行，在同一时间启动两个或两个以上的操作，大大提高计算性能。

空间上的并行：是指多个处理机并发的执行计算，即通过网络将两个以上的处理机连接起来，达到同时计算同一个任务的不同部分，或者单个处理机无法解决的大型问题。

12. 与 10.110.12.29mask 255.255.255.224 属于同一网段的主机 IP 地址有哪些？

根据你提供的 ip 地址和掩码计算出来的 ip 地址段为 10.110.12.0/27.
也就是从 10.110.12.0 到 10.110.12.31.

13. 讲一讲 Makefile 的内容.

target - 目标文件，可以是 Object File，也可以是可执行文件
prerequisites - 生成 target 所需要的文件或者目标
command - make 需要执行的命令（任意的 shell 命令），Makefile 中的命令必须以［tab］开头

显示规则 :: 说明如何生成一个或多个目标文件(包括 生成的文件，文件的依赖文件，生成的命令)
隐晦规则 :: make 的自动推导功能所执行的规则
变量定义 :: Makefile 中定义的变量
文件指示 :: Makefile 中引用其他 Makefile；指定 Makefile 中有效部分；定义一个多行命令
注释 :: Makefile 只有行注释 "#"，如果要使用或者输出"#"字符，需要进行转义，"\#"
最后，还值得一提的是，在 Makefile 中的命令，必须要以[Tab]键开始。

14. 讲一讲 C++的内联函数

内联函数 inline：引入内联函数的目的是为了解决程序中函数调用的效率问题，这么说吧，程序在编译器编译的时候，编译器将程序中出现的内联函数的调用表达式用内联函数的函数体进行替换，而对于其他的函数，都是在运行时候才被替代。这其实就是个空间代价换时间的 i 节省。所以内联函数一般都是 1-5 行的小函数。在使用内联函数时要留神：
1.在内联函数内不允许使用循环语句和开关语句；
2.内联函数的定义必须出现在内联函数第一次调用之前；
3.类结构中所在的类说明内部定义的函数是内联函数。

15. vector, deque, list, set, map 底层数据结构

vector（向量）——STL 中标准而安全的数组。只能在 vector 的“前面”增加数据。
deque（双端队列 double-ended queue）——在功能上和 vector 相似，但是可以在前后两端向其中添加数据。
list（列表）——游标一次只可以移动一步。如果你对链表已经很熟悉，那么 STL 中的 list

则是一个双向链表（每个节点有指向前驱和指向后继的两个指针）。

set（集合）——包含了经过排序了的数据，这些数据的值(value)必须是唯一的。

map （映射）——经过排序了的二元组的集合，map 中的每个元素都是由两个值组成，其中的 key（键值，一个 map 中的键值必须是唯一的）是在排序或搜索时使用，它的值可以在容器中重新获取；而另一个值是该元素关联的数值。比如，除了可以 ar[43] = "overripe"这样找到一个数据，map 还可以通过 ar["banana"] = "overripe"这样的方法找到一个数据。如果你想获得其中的元素信息，通过输入元素的全名就可以轻松实现。

16. 宏定义的优缺点

优点：

1. 提高了程序的可读性，同时也方便进行修改；
2. 提高程序的运行效率：使用带参的宏定义既可完成函数调用的功能，又能避免函数的出栈与入栈操作，减少系统开销，提高运行效率；
3. 宏是由预处理器处理的，通过字符串操作可以完成很多编译器无法实现的功能。比如##连接符。

缺点：

1. 由于是直接嵌入的，所以代码可能相对多一点；
2. 嵌套定义过多可能会影响程序的可读性，而且很容易出错；
3. 对带参的宏而言，由于是直接替换，并不会检查参数是否合法，存在安全隐患。

17. bfs 和 dfs 如何遍历

1.深度优先搜索（DFS）

原文里的深度优先搜索代码是有问题的，那是中序遍历的推广，而深度优先搜索是先序遍历的推广，我这里把两种代码都给出来，深度优先搜索的非递归实现使用了一个栈。

深度优先遍历图的方法是，从图中某顶点 v 出发：

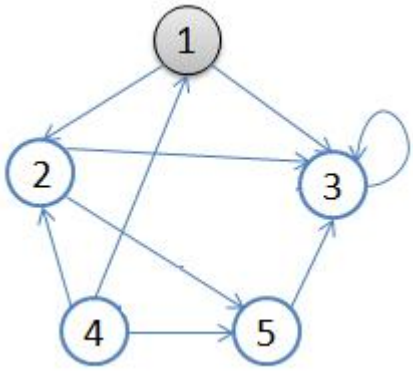
a.访问顶点 v；

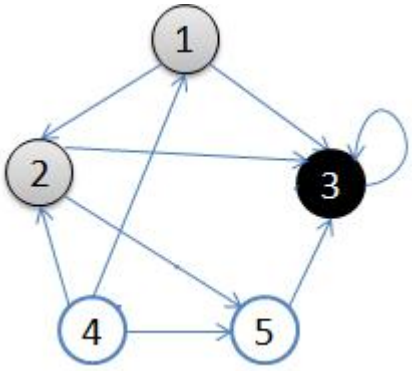
b.依次从 v 的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和 v 有路径相通的顶点都被访问；

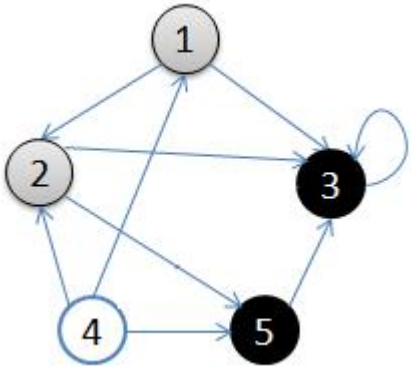
c.若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

用一副图来表达这个流程如下：

1).从 v = 顶点 1 开始出发，先访问顶点 1







2).按深度优先搜索递归访问 v 的某个未被访问的邻接点 2，顶点 2 结束后，应该访问 3 或 5 中的某一个，这里为顶点 3，此时顶点 3 不再有出度，因此回溯到顶点 2，再访问顶点 2 的另一个邻接点 5，由于顶点 5 的唯一一条边的弧头为 3，已经访问了，所以此时继续回溯到顶点 1，找顶点 1 的其他邻接点。

上图可以用邻接矩阵来表示为：

```
int maze[][] = {
    { 0, 1, 1, 0, 0 },
    { 0, 0, 1, 0, 1 },
    { 0, 0, 1, 0, 0 },
    { 1, 1, 0, 0, 1 },
    { 0, 0, 1, 0, 0 }
};
```

具体的代码如下:

```
import java.util.LinkedList;
import classEnhance.EnhanceModual;

public class DepthFirst extends EnhanceModual {

    @Override
    public void internalEntrance() {
        // TODO Auto-generated method stub
        int maze[][] = {
                { 0, 1, 1, 0, 0 },
                { 0, 0, 1, 0, 1 },
                { 0, 0, 1, 0, 0 },
                { 1, 1, 0, 0, 1 },
                { 0, 0, 1, 0, 0 }
        };
        dfs(maze, 1);
    }

    public void dfs(int[][] adjacentArr, int start) {
        int nodeNum = adjacentArr.length;
        if (start <= 0 || start > nodeNum || (nodeNum == 1 && start != 1)) {
            System.out.println("Wrong input ! ");
            return;
        } else if (nodeNum == 1 && start == 1) {
            System.out.println(adjacentArr[0][0]);
            return;
        }

        int[] visited = new int[nodeNum + 1];//0 表示结点尚未入栈，也未访问
        LinkedList<Integer> stack = new LinkedList<Integer>();
        stack.push(start);
        visited[start] = 1;//1 表示入栈

        while (!stack.isEmpty()) {
            int nodeIndex = stack.peek();
            boolean flag = false;
            if(visited[nodeIndex] != 2){
                System.out.println(nodeIndex);
                visited[nodeIndex] = 2;//2 表示结点被访问
            }
```

```
            //沿某一条路径走到无邻接点的顶点
            for (int i = 0; i < nodeNum; i++) {
                if (adjacentArr[nodeIndex - 1][i] == 1 &&
                    visited[i + 1] == 0) {
                    flag = true;
                    stack.push(i + 1);
                    visited[i + 1] = 1;
                    break;//这里的 break 不能掉！！！！
                }
            }

            //回溯
            if(!flag){
                int visitedNodeIndex = stack.pop();
            }

        }
    }
}
```

广度优先搜索（BFS）

广度优先搜索是按层来处理顶点，距离开始点最近的那些顶点首先被访问，而最远的那些顶点则最后被访问，这个和树的层序变量很像，BFS 的代码使用了一个队列。搜索步骤：

a .首先选择一个顶点作为起始顶点，并将其染成灰色，其余顶点为白色。

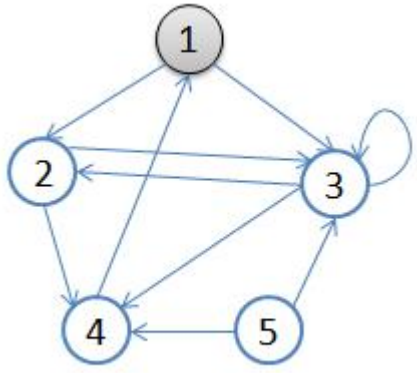
b. 将起始顶点放入队列中。

c. 从队列首部选出一个顶点，并找出所有与之邻接的顶点，将找到的邻接顶点放入队列尾部，将已访问过顶点涂成黑色，没访问过的顶点是白色。如果顶点的颜色是灰色，表示已经发现并且放入了队列，如果顶点的颜色是白色，表示还没有发现

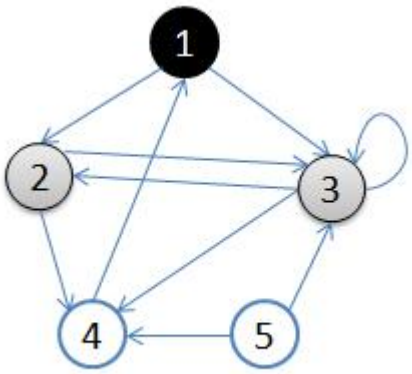
d. 按照同样的方法处理队列中的下一个顶点。

基本就是出队的顶点变成黑色，在队列里的是灰色，还没入队的是白色。

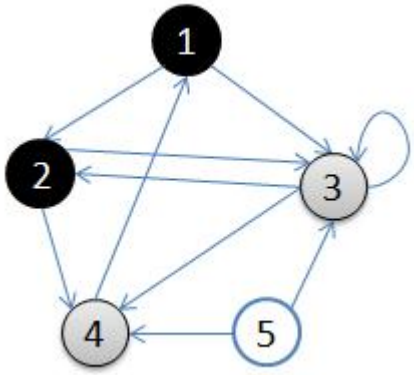
用一副图来表达这个流程如下：



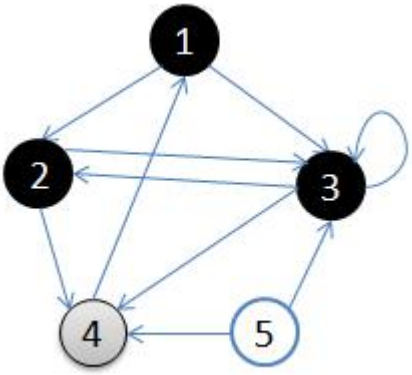
1. 初始状态，从顶点 1 开始，队列={1}



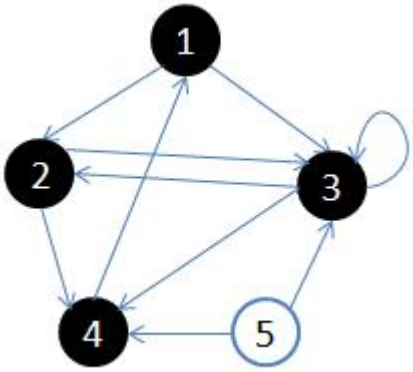
2. 访问 1 的邻接顶点，1 出队变黑，2,3 入队，队列={2,3,}



3. 访问 2 的邻接顶点，2 出队，4 入队，队列={3,4}



4. 访问 3 的邻接顶点，3 出队，队列={4}



5.访问 4 的邻接顶点，4 出队，队列={ 空}
分析：

从顶点 1 开始进行广度优先搜索：
初始状态，从顶点 1 开始，队列={1}
访问 1 的邻接顶点，1 出队变黑，2,3 入队，队列={2,3,}
访问 2 的邻接顶点，2 出队，4 入队，队列={3,4}
访问 3 的邻接顶点，3 出队，队列={4}
访问 4 的邻接顶点，4 出队，队列={ 空}
顶点 5 对于 1 来说不可达。

上面图可以用如下邻接矩阵来表示：

```
int maze[][] = {
    { 0, 1, 1, 0, 0 },
    { 0, 0, 1, 1, 0 },
    { 0, 1, 1, 1, 0 },
    { 1, 0, 0, 0, 0 },
    { 0, 0, 1, 1, 0 }
};
```

具体的代码如下，这段代码有两个功能，bfs（）函数求出从某顶点出发的搜索结果，minPath（）函数求从某一顶点出发到另一顶点的最短距离：

```
import java.util.LinkedList;
import classEnhance.EnhanceModual;

public class BreadthFirst extends EnhanceModual {

    @Override
    public void internalEntrance() {
        // TODO Auto-generated method stub
        int maze[][] = {
            { 0, 1, 1, 0, 0 },
```

```
            { 0, 0, 1, 1, 0 },
            { 0, 1, 1, 1, 0 },
            { 1, 0, 0, 0, 0 },
            { 0, 0, 1, 1, 0 }
        };

        bfs(maze, 5);//从顶点 5 开始搜索图

        int start = 5;
        int[] result = minPath(maze, start);
        for(int i = 1; i < result.length; i++){
            if(result[i] !=5 ){
                System.out.println("从顶点" + start +"到顶点" +
                    i + "的最短距离为：" + result[i]);
            }else{
                System.out.println("从顶点" + start +"到顶点" +
                    i + "不可达");
            }
        }
    }

    public void bfs(int[][] adjacentArr, int start) {
        int nodeNum = adjacentArr.length;
        if (start <= 0 || start > nodeNum || (nodeNum == 1 && start != 1)) {
            System.out.println("Wrong input ! ");
            return;
        } else if (nodeNum == 1 && start == 1) {
            System.out.println(adjacentArr[0][0]);
            return;
        }

        //0 表示顶点尚未入队，也未访问，注意这里位置 0 空出来了
        int[] visited = new int[nodeNum + 1];
        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.offer(start);
        visited[start] = 1;//1 表示入队

        while (!queue.isEmpty()) {
            int nodeIndex = queue.poll();
            System.out.println(nodeIndex);
            visited[nodeIndex] = 2;//2 表示顶点被访问

            for (int i = 0; i < nodeNum; i++) {
                if (adjacentArr[nodeIndex - 1][i] == 1 &&
```

```
                visited[i + 1] == 0) {
                queue.offer(i + 1);
                visited[i + 1] = 1;
            }
        }
    }
}

/*
 * 从 start 顶点出发，到图里各个顶点的最短路径
 */
public int[] minPath(int[][] adjacentArr, int start) {

    int nodeNum = adjacentArr.length;

    LinkedList<Integer> queue = new LinkedList<Integer>();
    queue.offer(start);
    int path = 0;
    int[] nodePath = new int[nodeNum + 1];
    for (int i = 0; i < nodePath.length; i++) {
        nodePath[i] = nodeNum;
    }
    nodePath[start] = 0;

    int incount = 1;
    int outcount = 0;
    int tempcount = 0;

    while (path < nodeNum) {
        path++;
        while (incount > outcount) {
            int nodeIndex = queue.poll();
            outcount++;

            for (int i = 0; i < nodeNum; i++) {
                if (adjacentArr[nodeIndex - 1][i] == 1 &&
                    nodePath[i + 1] == nodeNum) {
                    queue.offer(i + 1);
                    tempcount++;
                    nodePath[i + 1] = path;
                }
            }
        }
```

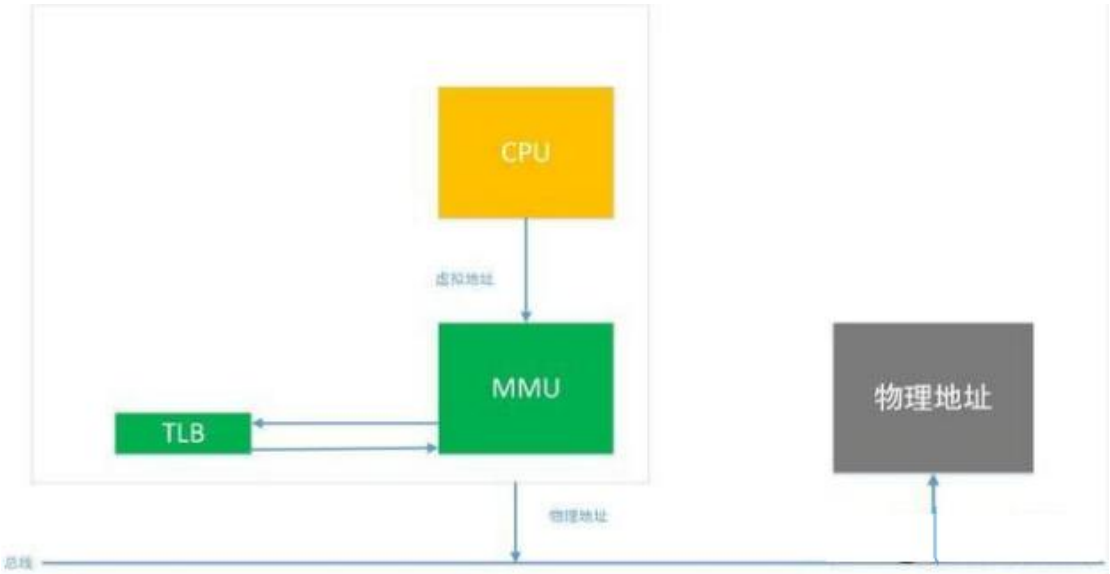
```
                incount = tempcount;
                tempcount = 0;
                outcount = 0;
            }

            return nodePath;
        }
    }
```

18. a[1]+a[2]+..+a[n]如何进行并行设计

心乘以 2.

19. CPU 如果访问内存?

通过内存管理单元(MMU)
先看一张简单的 CPU 访问内存的流程图：



TLB：转换 lookaside 缓存，有了它可以让虚拟地址到物理地址转换速度大增。
从上图中可以清楚的知道了,CPU,DDR,MMU 它们三者之间的关系。CPU 在 MMU 开启的情况下，访问的都是虚拟地址。
首先通过 MMU 将虚拟地址转换为物理地址，
然后再通过总线上去访问内存(我们都知道内存是挂在总线上的)。
那 MMU 是怎么将虚拟地址转换为物理地址呢?当然是通过页表的方式。MMU 从页表中查出虚拟地址对应的物理地址是什么，然后就去访问物理内存了。

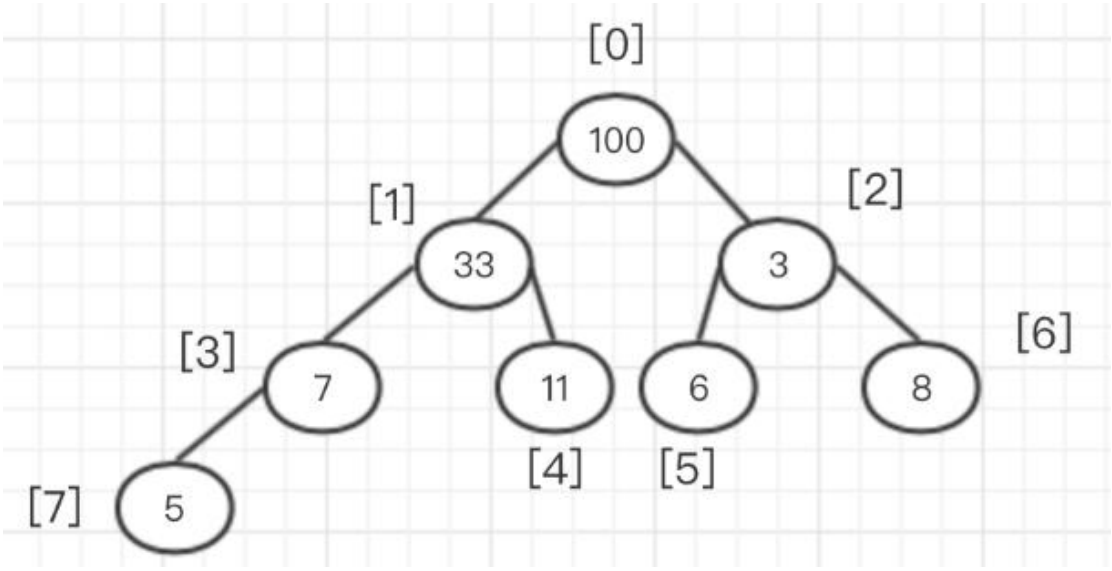
20. 找出在 A 数组中，B 数组中没有的数字，在 B 数组中，A 数组中没有的数字

```
public static void  find(int arr[],int[]b){
        HashMap<Integer,Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < arr.length; i++) {
            map.put(arr[i],0);
        }
        for (int i=0;i<b.length;i++) {
            if(map.containsKey(b[i])){
                map.put(b[i],1);
            }else{
                System.out.println("在 B 数组中 A 不存在的数字");
                System.out.println(b[i]);
            }
        }
        for (int i = 0; i <arr.length ; i++) {
            if(map.get(arr[i])==0){
                System.out.println("在 A 数组中存在的,在 B 数组不存在的数字");
                System.out.println(arr[i]);
            }
        }
}
```

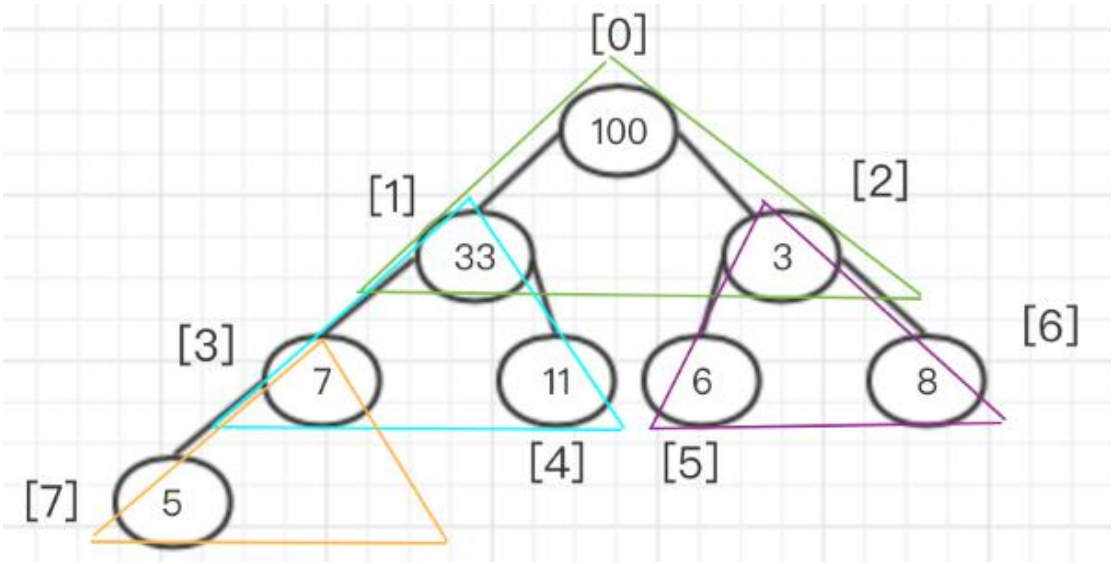
21. 堆排序的思路

堆是一个完全二叉树
完全二叉树即是：若设二叉树的深度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。
堆满足两个性质：堆的每一个父节点数值都大于（或小于）其子节点，堆的每个左子树和右子树也是一个堆。
堆分为最小堆和最大堆。最大堆就是每个父节点的数值要大于孩子节点，最小堆就是每个父节点的数值要小于孩子节点。排序要求从小到大的话，我们需要建立最大堆，反之建立最小堆。
堆的存储一般用数组来实现。假如父节点的数组下标为 i 的话，那么其左右节点的下标分别为：(2*i+1)和 (2*i+2)。如果孩子节点的下标为 j 的话，那么其父节点的下标为 (j-1)/2。
完全二叉树中，假如有 n 个元素，那么在堆中最后一个父节点位置为(n/2-1)。
算法思想
建立堆
调整堆
交换堆顶元素和堆的最后一个元素

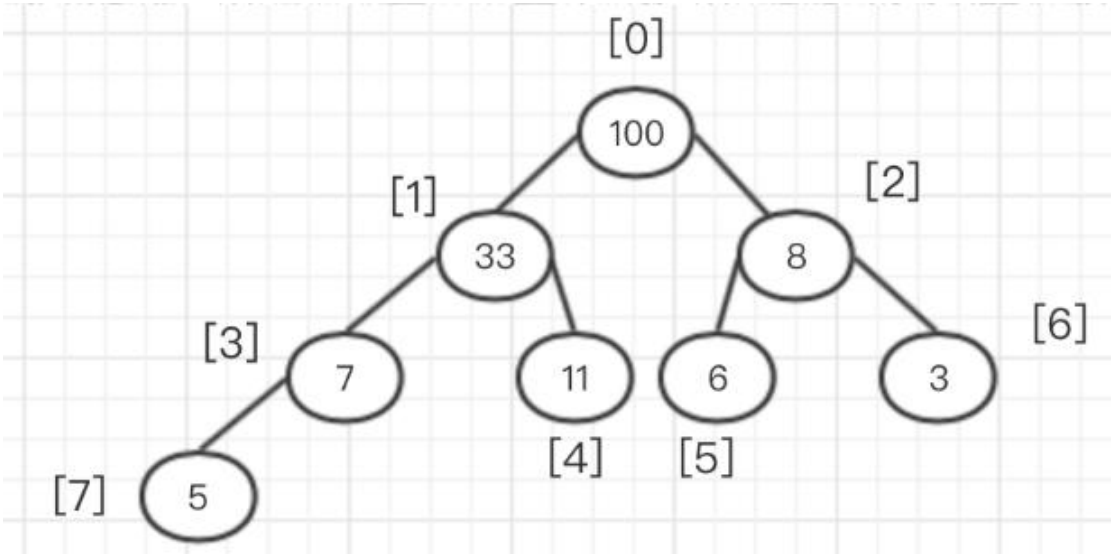
假如现在有一个数组 a[8]={100,33,3,7,11,6,8,5};首先我们要建立完全二叉树。如下图所示：



然后根据各个父节点，进行一个划分。如图所示：

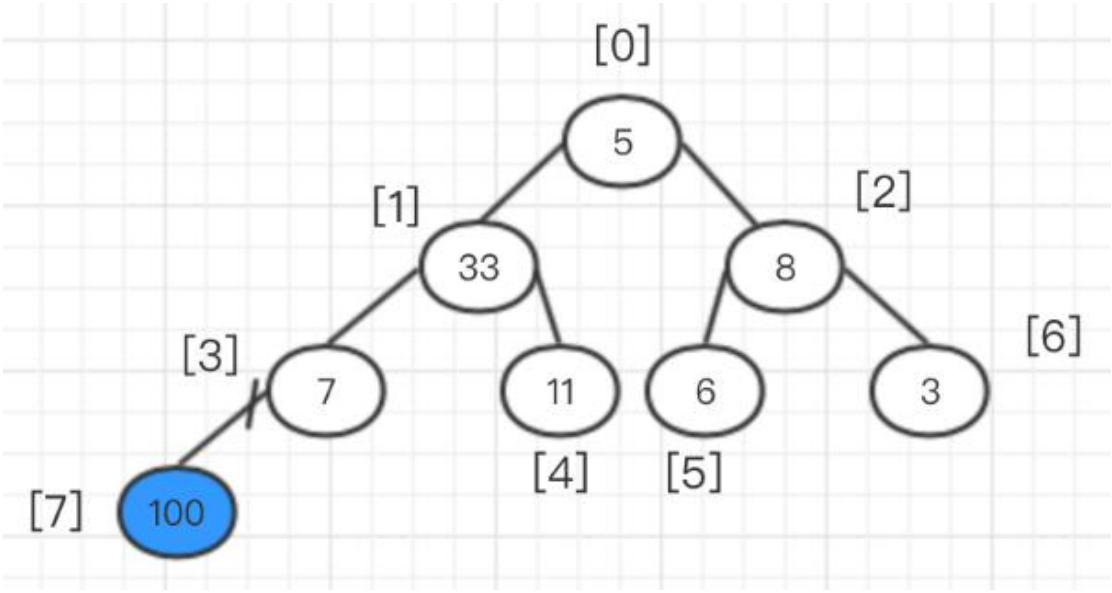


该算法的就是将各个父节点与其自己的孩子结点进行对比，然后交换的过程。根据例子，建立一个最大堆，要求每一个父节点的数值是大于孩子节点的。顺序是从最后一个父节点开始（从左至右，从下至上）。所以第一个父节点是数组下标为 3 的 7，黄色区域中的父节点与孩子节点对比后，发现父节点已经大于孩子节点了，所以不用进行交换了。接下来的顺序就是紫色框中的二叉树，就是数组下标为 2 的数字 3，以数字 3 为父节点，对比它的孩子节点，从左到右，发现右孩子的数值比父节点大，那么将右节点和父节点进行数值交换。交换后的堆如图所示：

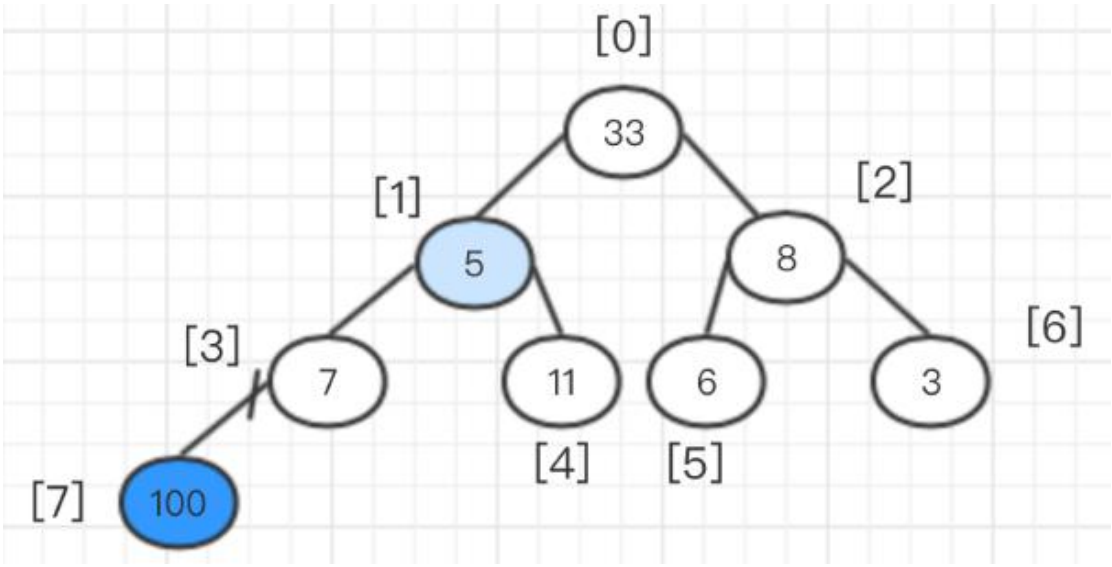


交换结束后，再看蓝色框中的二叉树，就是以数组下标为 1 的数字 33，以 33 为父节点，看它的孩子节点是否大于父节点，结果发现不大于，则不用交换。此时这个完全二叉树已经是一个堆。

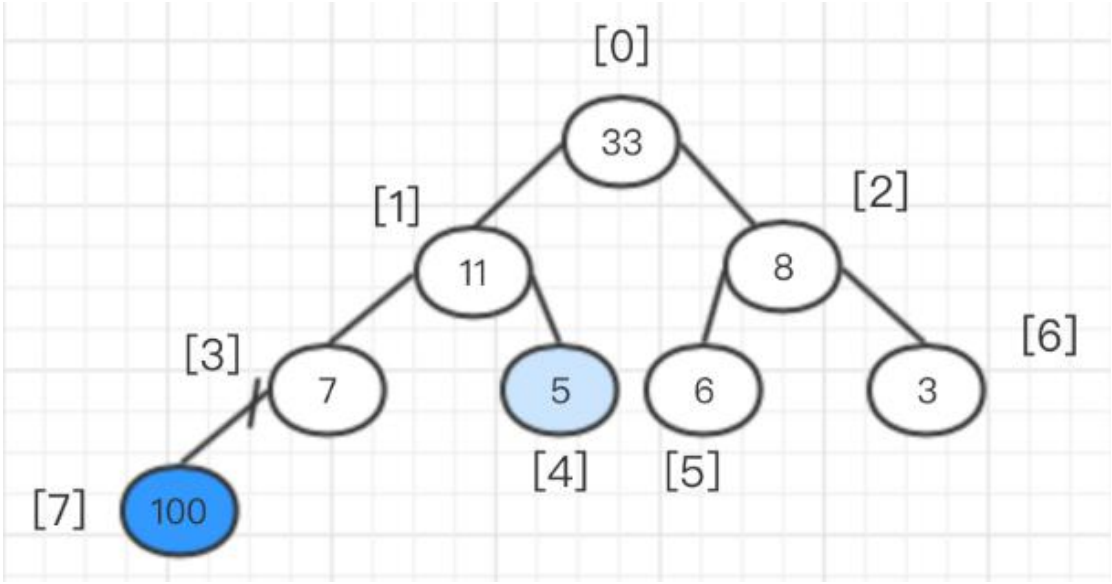
接下来可以讲堆顶元素和堆的最后一个元素进行互换，然后再降最后一个元素至于完全二叉树外。



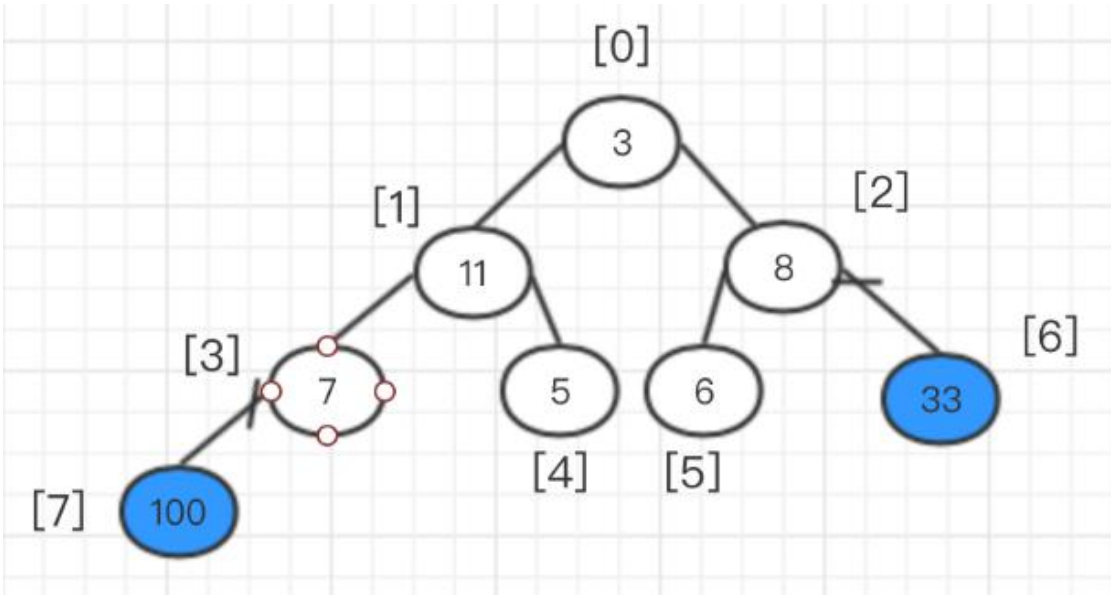
此时完全二叉树中，除去 100 这个元素之后，这个完全二叉树已经不满足堆的性质，所以要进行调整，此时的每次调整要从根节点（和创建堆不一样）开始进行调整，而且父节点和孩子节点交换后，要追踪到交换的孩子节点上（我用浅蓝色标志的部分）。



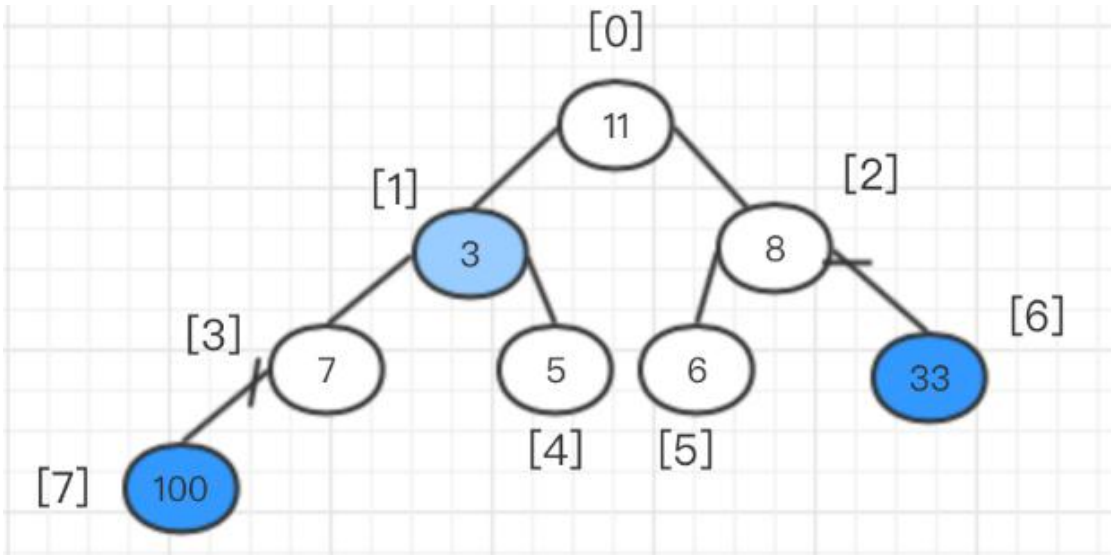
此时 33 和 5 交换了位置，那么就要从 5 为父节点，开始往下再和孩子节点进行比较，此时发现孩子节点和父节点中最大为 11，那么将 11 和父节点（即数字 5）互换位置。

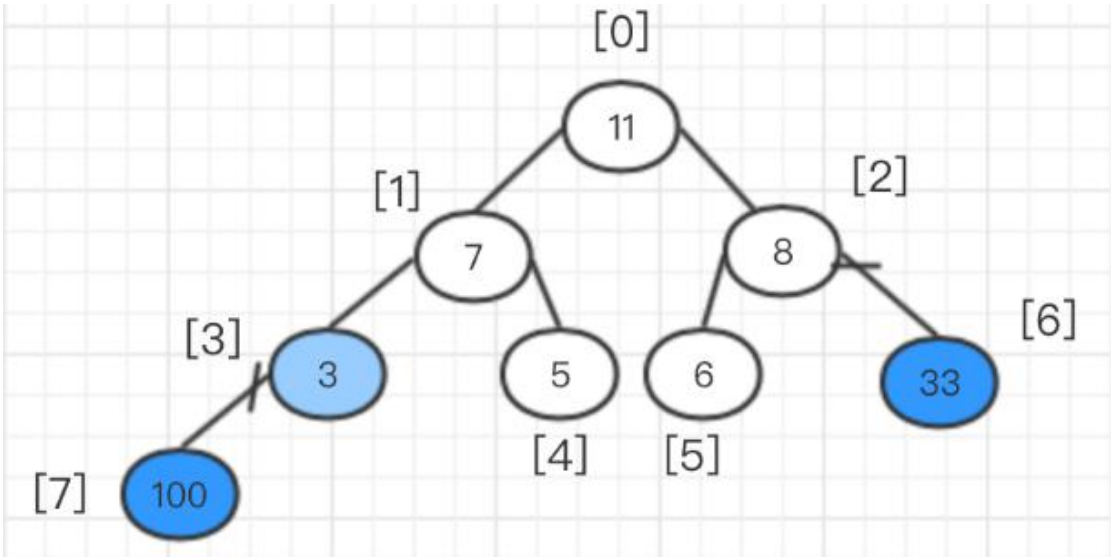


此时堆已经调整好，再将堆顶元素和堆的最后一个元素互换，即将 33 和 3 互换，然后再降 33 至于完全二叉树外。

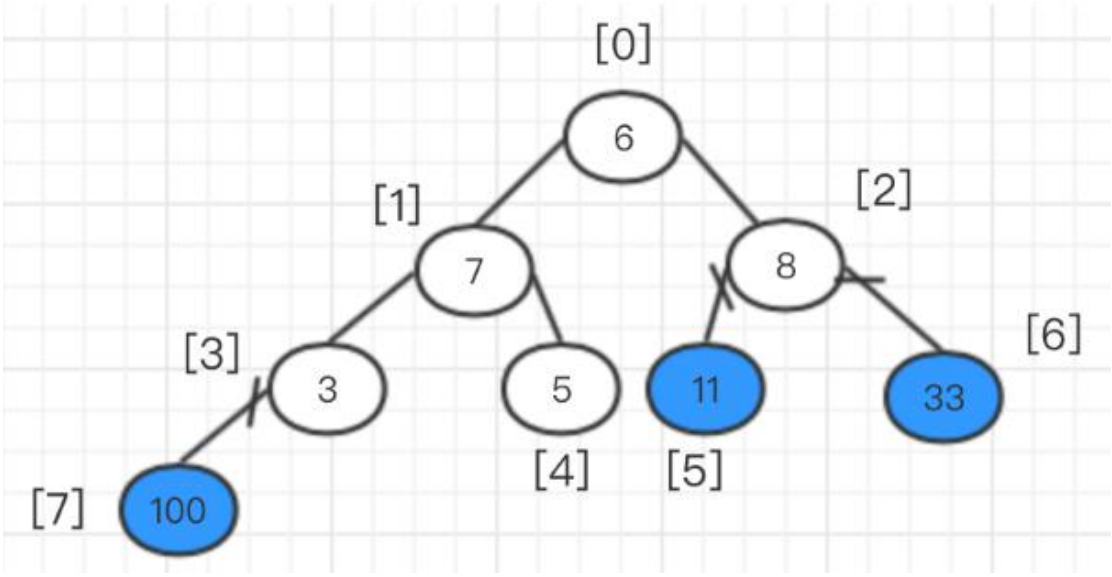


此时再进行堆的调整，从根开始。

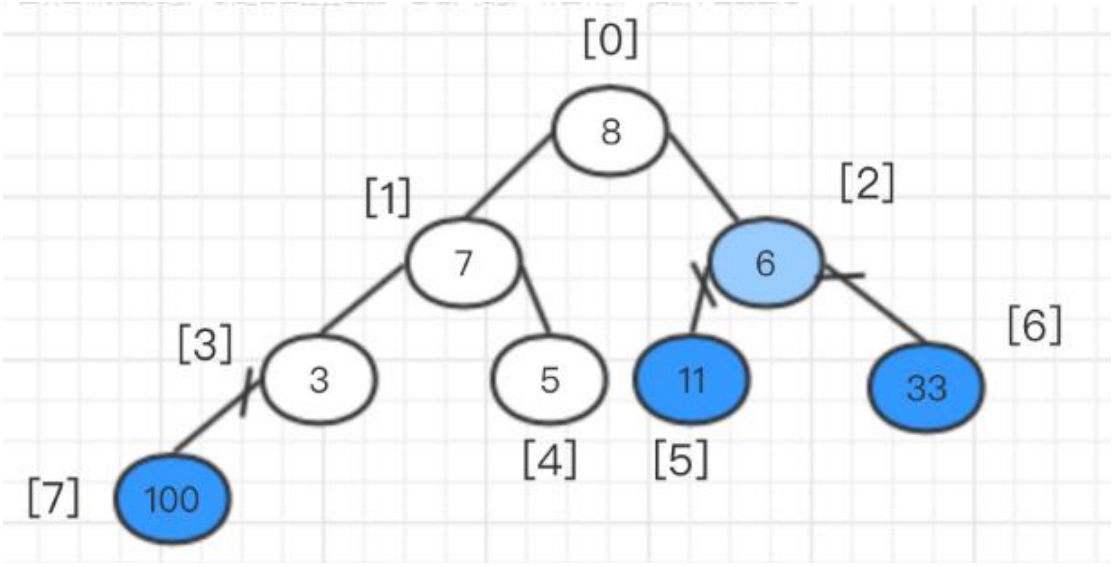




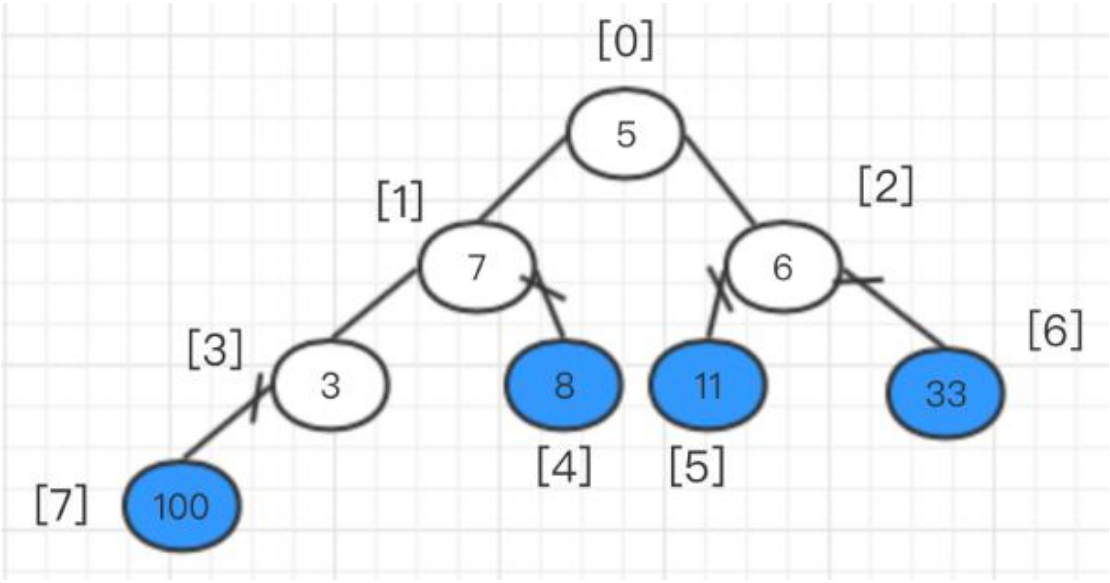
此时调整完堆后，再将堆顶元素和最后一个元素互换位置，即将 11 和 6 互换，再将 11 至于完全二叉树外。



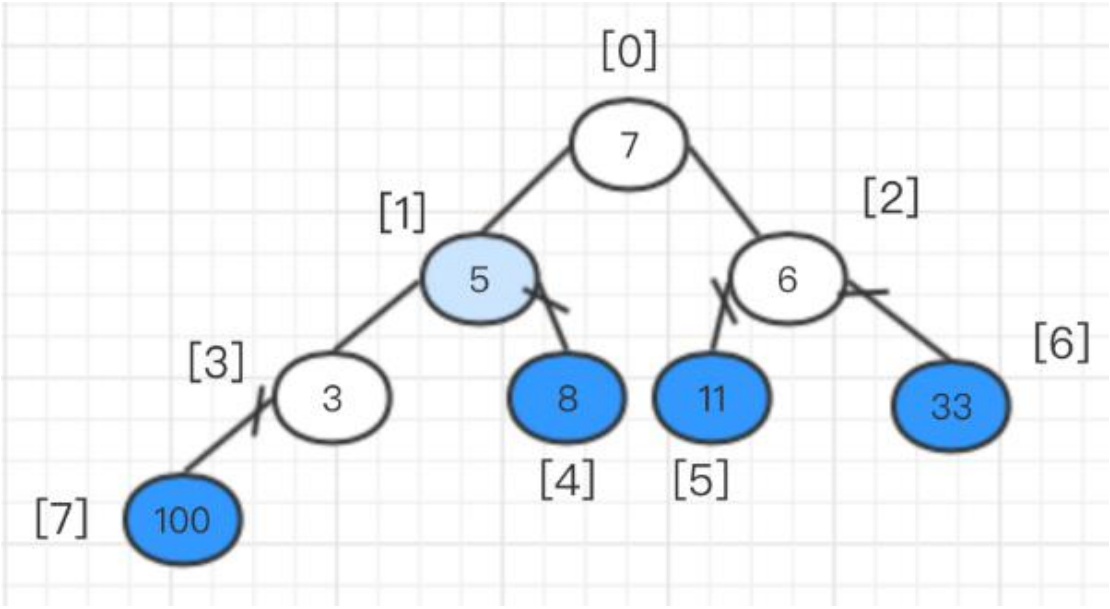
此时再进行完全二叉树的调整，即就是白色填充的元素进行调整。从根开始，按照上面的方法。



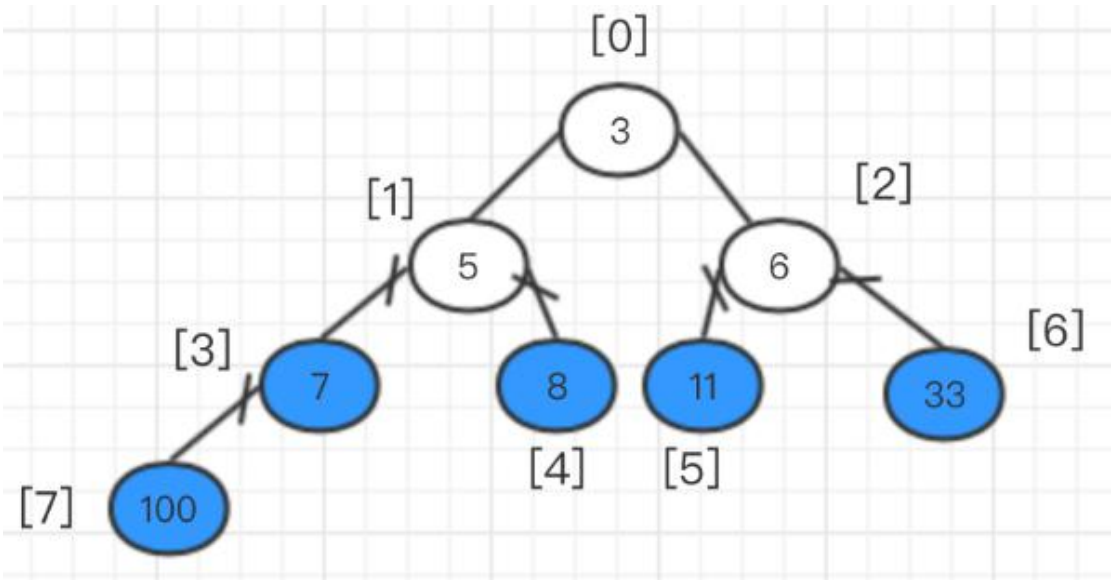
此时又是一个调整好的堆，将堆顶元素和最后一个元素互换。



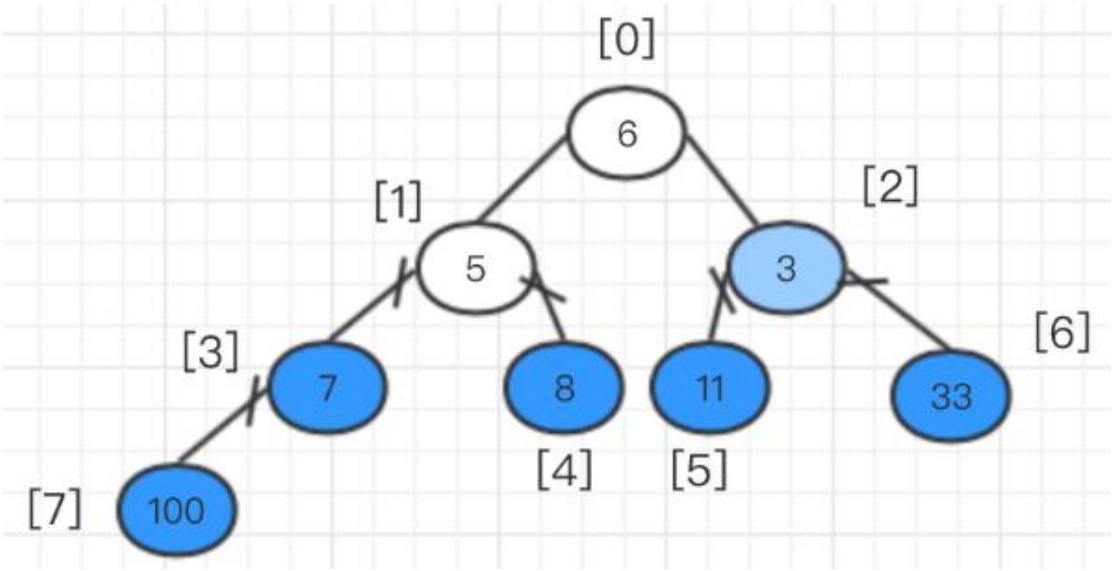
再进行完全二叉树的调整



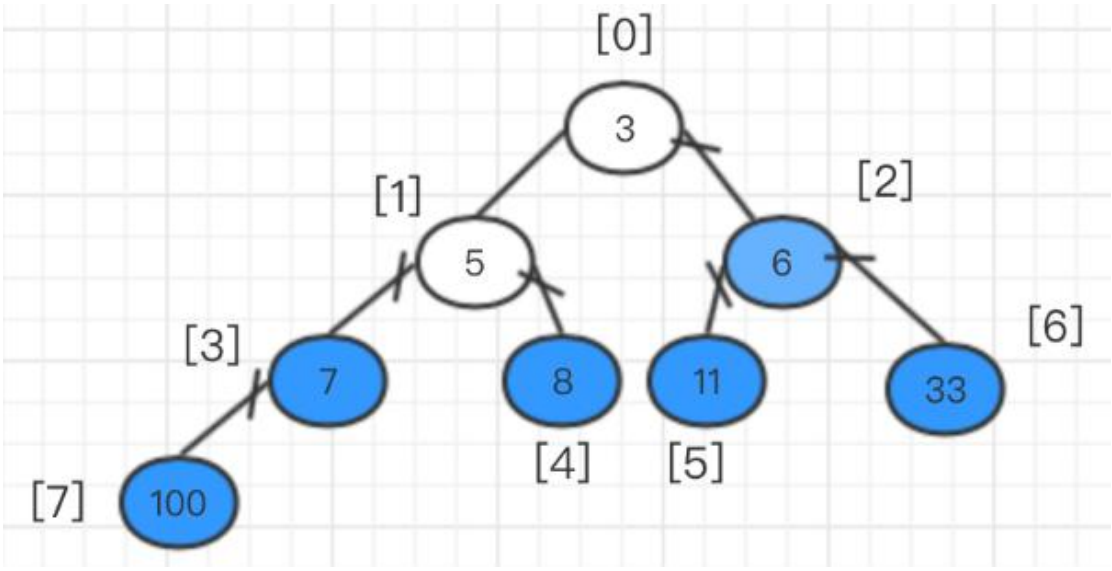
此时再进行堆顶元素和最后一个元素的互换



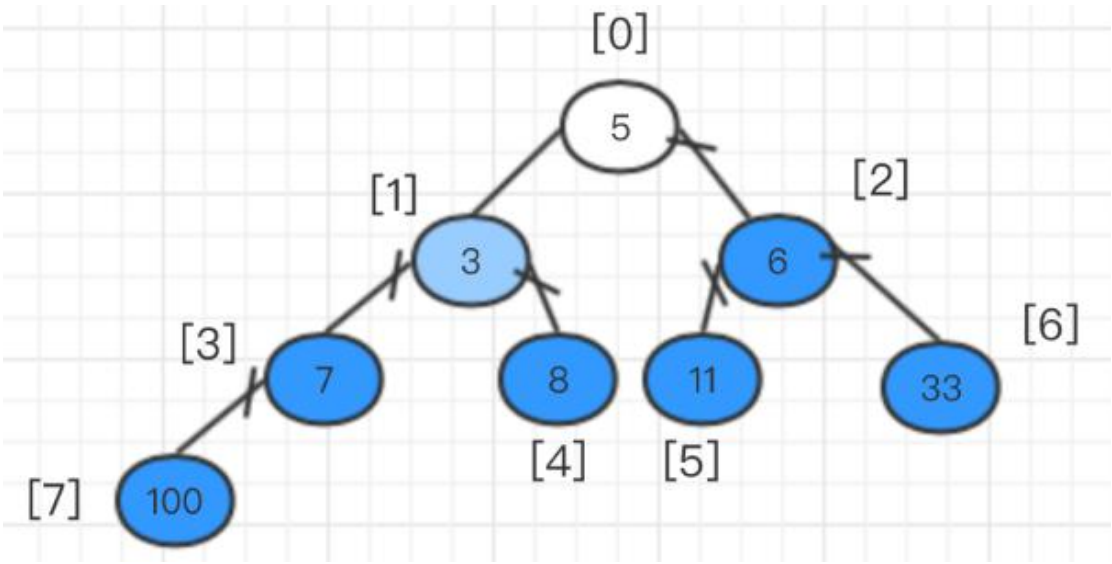
再进行完全二叉树的调整



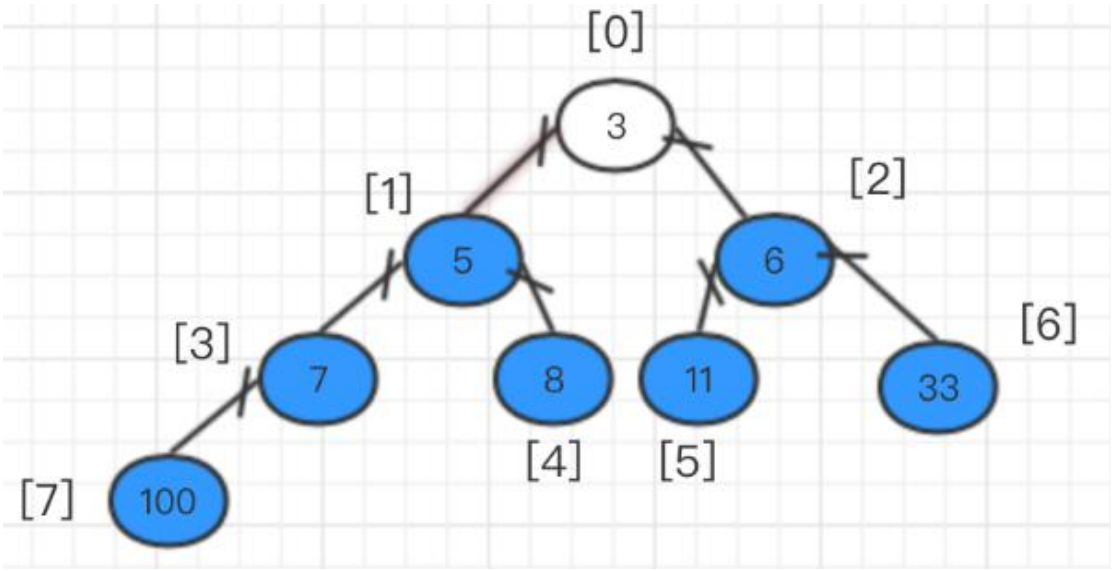
最后进行堆顶元素的互换：



最后一次调整：



最后一次堆顶元素交换：



此时数组已经是一个有序数组，是一个生序数组，用 for 循环输出即可。

22. Linux 查看文件大小命令.

1. 使用 stat 命令查看
stat 命令一般用于查看文件的状态信息。stat 命令的输出信息比 ls 命令的输出信息要更详细。
2. 使用 wc 命令
wc 命令一般用于统计文件的信息，比如文本的行数，文件所占的字节数。
3. 使用 du 命令
du 命令一般用于统计文件和目录所占用的空间大小。
4. 使用 ls 命令
ls 命令一般用于查看文件和目录的信息，包括文件和目录权限、拥有者、所对应的组、文件大小、修改时间、文件对应的路径等等信息。
5. 使用 ll 命令(其实就是 ls -l 的别名)
在大部分的 Linux 系统中，都已经设置了 ls -l 的别名为 ll，所以并不存在 ll 的命令，ll 只是一个别名命令而已。
Linux 使用 ll 命令查看文件大小

23. 布隆过滤算法

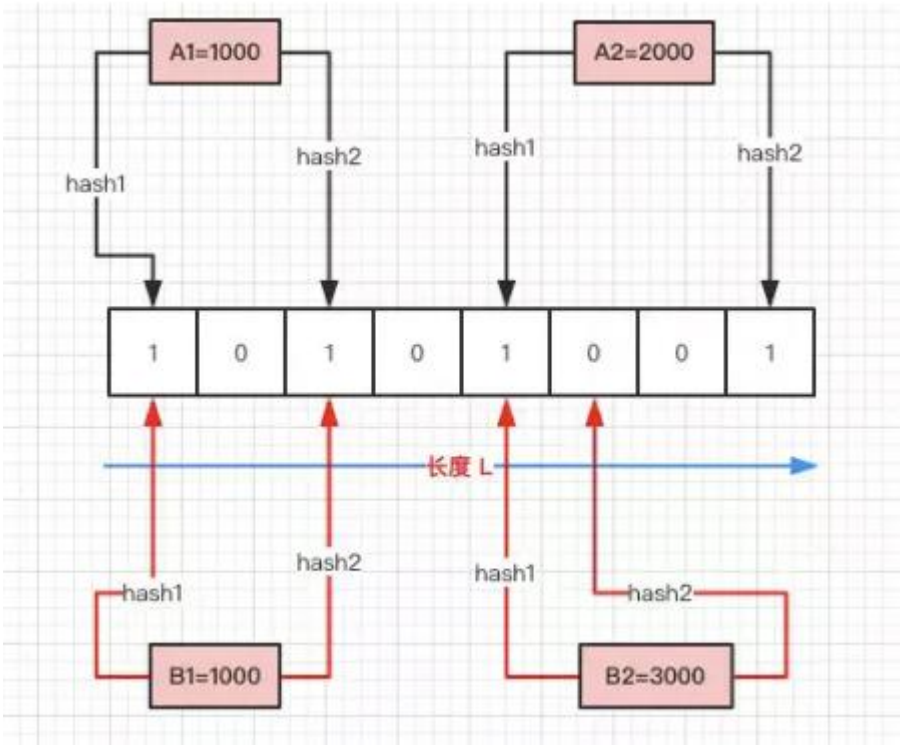
布隆过滤器使用二进制向量结合 hash 函数来记录任意一条数据是否已经存在于集合中。
布隆过滤器的执行流程为：
首先申请包含 SIZE 个 bit 位的 Bit 集合，并将所有 Bit 置 0。
然后使用数种（k）不同的哈希函数对目标数据进行哈希计算并得到 k 个哈希值（确保哈希值不超过 SIZE 大小），然后将 Bit 集合中以哈希值为下标所处的 bit 值置为 1，

由于使用了 k 个哈希函数，因此记录一条数据的信息将在 Bit 集合中把 k 个 bit 值置为 1。

由于哈希函数的稳定性，任意两条相同的数据在 Bit 集合中所对应的 k 个 bit 位置是完全相同的。那么在检测某一条数据是否已经在 Bit 集合中有记录时，只需检测该条数据的 k 个哈希值在 Bit 集合中对应的位置的 bit 是否均已被标记为 1，相反的只要其存在一个哈希值对应的 bit 位置未被标记为 1，则证明该值未被记录过。

使用示例

布隆过滤器的示例如下：



大体过程为：

首先初始化一个二进制的数组，长度设为 L（图中为 8），同时初始值全为 0 。

当写入一个 A1=1000 的数据时，需要进行 H 次 hash 函数的运算（这里为 2 次）；与 HashMap 有点类似，通过算出的 HashCode 与 L 取模后定位到 0、2 处，将该处的值设为 1。

A2=2000 也是同理计算后将 4、7 位置设为 1。

当有一个 B1=1000 需要判断是否存在时，也是做两次 Hash 运算，定位到 0、2 处，此时他们的值都为 1 ，所以认为 B1=1000 存在于集合中。

当有一个 B2=3000 时，也是同理。第一次 Hash 定位到 index=4 时，数组中的值为 1，所以再进行第二次 Hash 运算，结果定位到 index=5 的值为 0，所以认为 B2=3000 不存在于集合中。

优缺点

优点

时间复杂度为 O(n)，且布隆过滤器不需要存储元素本身，使用位阵列，占用空间也很小。

缺点

通过布隆过滤，我们能够准确判断一个数不存在于某个集合中，但对于存在于集合中这个结论，布隆过滤会有误报（可能存在两组不同数据但其多个哈希值完全一样的情况）。但是通过控制 Bit 集合的大小（即 SIZE）以及哈希函数的个数，可以将出现冲突的概率控制在极小的范围内，或者通过额外建立白名单的方式彻底解决哈希冲突问题。
误判率计算公式为(1 - e(-nk/SIZE))k，其中 n 为目标数据的数量，SIZE 为 Bit 集合大小，k 为使用的哈希函数个数；假设现有一千万条待处理数据，Bit 集合大小为 2^30（约 10 亿，即占用内存 128MB），使用 9 个不同的哈希函数，计算可得任意两条数据其 9 次哈希得到的哈希值均相同（不考虑顺序）的概率为 2.6e-10，约为 38 亿分之一。

24. 给定一组非负整数组成的数组 h，代表一组柱状图的高度，其中每个柱子的宽度都为 1。 在这组柱状图中找到能组成的最大矩形的面积。 入参 h 为一个整型数组，代表每个柱子的高度，返回面积的值。

输入描述:
输入包括两行,第一行包含一个整数 n(1 ≤ n ≤ 10000)
第二行包括 n 个整数,表示 h 数组中的每个值,h_i(1 ≤ h_i ≤ 1,000,000)
输出描述:
输出一个整数,表示最大的矩阵面积。

输入例子 1:
6
2 1 5 6 2 3
输出例子 1:
10

```
#include <bits/stdc++.h>
using namespace std;

const int maxn=10000+5;
typedef long long ll;
int h[maxn],l[maxn],r[maxn];

int main()
{
    //freopen("in.txt","r",stdin);
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
        scanf("%d",&h[i]);
    l[1]=0;
```

```
    for(int i=2;i<=n;i++)
    {
        int lp=i-1;
        while(h[lp]>=h[i])
            lp=l[lp];
        l[i]=lp;
    }
    r[n]=n+1;
    for(int i=n-1;i>=1;i--)
    {
        int hp=i+1;
        while(h[hp]>=h[i])
            hp=r[hp];
        r[i]=hp;
    }
    ll ans=0;
    for(int i=1;i<=n;i++)
        ans=max(ans,(ll)h[i]*(r[i]-l[i]-1));
    printf("%lld\n",ans);
    return 0;
}
```

25. 一个 C++源文件从文本到可执行文件经历的过程？

对于 C++源文件，从文本到可执行文件一般需要四个过程：
预处理阶段：对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。
编译阶段：将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件
汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件
链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

26. 请问 malloc 的原理，brk 系统调用和 mmap 系统调用的作用分别是什么？

malloc 函数用于动态分配内存。为了减少内存碎片和系统调用的开销，malloc 其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作为内存管理的基本单位。当用户申请内存时，直接从堆区分配一块合适的空闲块。malloc 采用隐式链表结构将堆区分成连续的、大小不一的块，包含已分配块和未分配块；同时 malloc 采用显示链表结构来管理所有的空闲块，即使用一个双向链表将空闲块连接起来，每一个空闲块记录了一个连续的、未分配的地址。

当进行内存分配时，malloc 会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc 采用边界标记法，根据每个块的前后块是否已经分配来决定是否进行块合并。

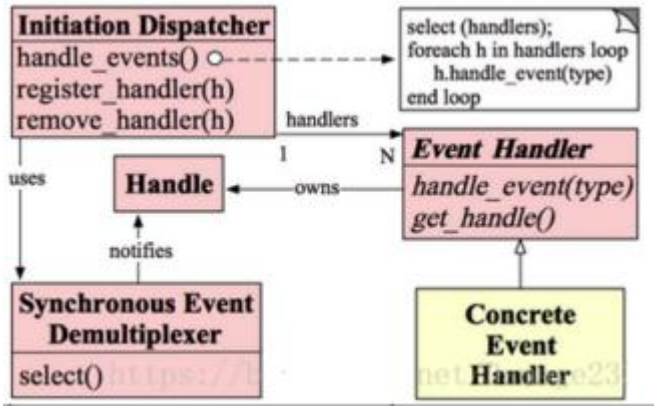
malloc 在申请内存时，一般会通过 brk 或者 mmap 系统调用进行申请。其中当申请内存小于 128K 时，会使用系统函数 brk 在堆区中分配；而当申请内存大于 128K 时，会使用系统函数 mmap 在映射区分配。

27. 请问共享内存相关 api?

Linux 允许不同进程访问同一个逻辑内存，提供了一组 API，头文件在 sys/shm.h 中。
1）新建共享内存 shmget
int shmget(key_t key,size_t size,int shmflg);
key：共享内存键值，可以理解为共享内存的唯一性标记。
size：共享内存大小
shmflag：创建进程和其他进程的读写权限标识。
返回值：相应的共享内存标识符，失败返回-1
2）连接共享内存到当前进程的地址空间 shmat
void *shmat(int shm_id,const void *shm_addr,int shmflg);
shm_id：共享内存标识符
shm_addr：指定共享内存连接到当前进程的地址，通常为 0，表示由系统来选择。
shmflg：标志位
返回值：指向共享内存第一个字节的指针，失败返回-1
3）当前进程分离共享内存 shmdt
int shmdt(const void *shmaddr);
4）控制共享内存 shmctl
和信号量的 semctl 函数类似，控制共享内存
int shmctl(int shm_id,int command,struct shmid_ds *buf);
shm_id：共享内存标识符
command: 有三个值
IPC_STAT:获取共享内存的状态，把共享内存的 shmid_ds 结构复制到 buf 中。
IPC_SET:设置共享内存的状态，把 buf 复制到共享内存的 shmid_ds 结构。
IPC_RMID:删除共享内存
buf：共享内存管理结构体。

28. 请问 reactor 模型组成.

reactor 模型要求主线程只负责监听文件描述上是否有事件发生，有的话就立即将该事件通知工作线程，除此之外，主线程不做任何其他实质性的工作，读写数据、接受新的连接以及处理客户请求均在工作线程中完成。其模型组成如下：



1）Handle：即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer 等。由于 Reactor 模式一般使用在网络编程中，因而这里一般指 Socket Handle，即一个网络连接。

2）Synchronous Event Demultiplexer（同步事件复用器）：阻塞等待一系列的 Handle 中的事件到来，如果阻塞等待返回，即表示在返回的 Handle 中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的 select 来实现。

3）Initiation Dispatcher：用于管理 Event Handler，即 EventHandler 的容器，用以注册、移除 EventHandler 等；另外，它还作为 Reactor 模式的入口调用 Synchronous Event Demultiplexer 的 select 方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的 Handle 将其分发给对应的 Event Handler 处理，即回调 EventHandler 中的 handle_event()方法。

4）Event Handler：定义事件处理方法：handle_event()，以供 InitiationDispatcher 回调使用。

5）Concrete Event Handler：事件 EventHandler 接口，实现特定事件处理逻辑。

29. 请问 Linux 虚拟地址空间

为了防止不同进程同一时刻在物理内存中运行而对物理内存的争夺和践踏，采用了虚拟内存。

虚拟内存技术使得不同进程在运行过程中，它所看到的是自己独自占有了当前系统的 4G 内存。所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。 事实上，在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，具体就是初始化进程控制表中内存相关的链表，实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data 段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射就好（叫做存储器映射），等到运行到对应的程序时，才会通过缺页异常，来拷贝数据。还有进程运行过程中，要动态分配内存，比如 malloc 时，也只是分配了虚拟内存，即为这块虚拟内存对应的页表项做相应设置，当进程真正访问到此数据时，才引发缺页异常。

请求分页系统、请求分段系统和请求段页式系统都是针对虚拟内存的，通过请求实现内存与外存的信息置换。

虚拟内存的好处：

1.扩大地址空间；

2.内存保护：每个进程运行在各自的虚拟内存地址空间，互相不能干扰对方。虚存还对特定的内存地址提供写保护，可以防止代码或数据被恶意篡改。
3.公平内存分配。采用了虚存之后，每个进程都相当于有同样大小的虚存空间。
4.当进程通信时，可采用虚存共享的方式实现。
5.当不同的进程使用同样的代码时，比如库文件中的代码，物理内存中可以只存储一份这样的代码，不同的进程只需要把自己的虚拟内存映射过去就可以了，节省内存
6.虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中。当一个程序等待它的一部分读入内存时，可以把 CPU 交给另一个进程使用。在内存中可以保留多个进程，系统并发度提高
7.在程序需要分配连续的内存空间的时候，只需要在虚拟内存空间分配连续空间，而不需要实际物理内存的连续空间，可以利用碎片

虚拟内存的代价：
1.虚存的管理需要建立很多数据结构，这些数据结构要占用额外的内存
2.虚拟地址到物理地址的转换，增加了指令的执行时间。
3.页面的换入换出需要磁盘 I/O，这是很耗时的
4.如果一页中只有一部分数据，会浪费内存。

30. 请问操作系统中的缺页中断

malloc()和 mmap()等内存分配函数，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页异常。
缺页中断：在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。每当所要访问的页面不在内存是，会产生一次缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。

缺页本身是一种中断，与一般的中断一样，需要经过 4 个处理步骤：
1、保护 CPU 现场
2、分析中断原因
3、转入缺页中断处理程序进行处理
4、恢复 CPU 现场，继续执行

但是缺页中断是由于所要访问的页面不存在于内存时，由硬件所产生的一种特殊的中断，因此，与一般的中断存在区别：
1、在指令执行期间产生和处理缺页中断信号
2、一条指令在执行期间，可能产生多次缺页中断
3、缺页中断返回是，执行产生中断的一条指令，而一般的中断返回是，执行下一条指令。

31. 请问 fork 和 vfork 的区别?

fork 的基础知识：

fork:创建一个和当前进程映像一样的进程可以通过 fork( )系统调用：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

成功调用 fork( )会创建一个新的进程，它几乎与调用 fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的 fork( )调用会返回 0。在父进程中 fork( )返回子进程的 pid。如果出现错误，fork( )返回一个负值。

最常见的 fork( )用法是创建一个新的进程，然后使用 exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的 Unix 系统中，创建进程比较原始。当调用 fork 时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的 Unix 系统采取了更多的优化，例如 Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

vfork 的基础知识：

在实现写时复制之前，Unix 的设计者们就一直很关注在 fork 后立刻执行 exec 所造成的地址空间的浪费。BSD 的开发者们在 3.0 的 BSD 系统中引入了 vfork( )系统调用。

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
```

除了子进程必须要立刻执行一次对 exec 的系统调用，或者调用_exit( )退出，对 vfork( )的成功调用所产生的结果和 fork( )是一样的。vfork( )会挂起父进程直到子进程终止或者运行了一个新的可执行文件的映像。通过这样的方式，vfork( )避免了地址空间的按页复制。在这个过程中，父进程和子进程共享相同的地址空间和页表项。实际上 vfork( )只完成了一件事：复制内部的内核数据结构。因此，子进程也就不能修改地址空间中的任何内存。

vfork( )是一个历史遗留产物，Linux 本不应该实现它。需要注意的是，即使增加了写时复制，vfork( )也要比 fork( )快，因为它没有进行页表项的复制。然而，写时复制的出现减少了对于替换 fork( )争论。实际上，直到 2.2.0 内核，vfork( )只是一个封装过的 fork( )。因为对 vfork( )的需求要小于 fork( )，所以 vfork( )的这种实现方式是可行的。

补充：写时复制

Linux 采用了写时复制的方法，以减少 fork 时对父进程空间进程整体复制带来的开销。写时复制是一种采取了惰性优化方法来避免复制时的系统开销。它的前提很简单：如果有多个进程要读取它们自己的那部门资源的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”，就存在着这样的幻觉：每个进程好像独占那个资源。从而就避免了复制带来的负担。如果一个进程要修改自己的那份资源“副本”，那么就会复制那份资源，并把复制的那份提供给进程。不过其中的复制对进程来说是透明的。这个进程就可以修改复制后的资源了，同时其他的进程仍然共享那份没有修改过的资源。所以这就是名称的由来：在写入时进行复制。

写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。
在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在 fork()调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。
写时复制在内核中的实现非常简单。与内核页相关的数据结构可以被标记为只读和写时复制。如果有进程试图修改一个页，就会产生一个缺页中断。内核处理缺页中断的方式就是对该页进行一次透明复制。这时会清除页面的 COW 属性，表示着它不再被共享。
现代的计算机系统结构中都在内存管理单元（MMU）提供了硬件级别的写时复制支持，所以实现是很容易的。
在调用 fork()时，写时复制是有很大优势的。因为大量的 fork 之后都会跟着执行 exec，那么复制整个父进程地址空间中的内容到子进程的地址空间完全是在浪费时间：如果子进程立刻执行一个新的二进制可执行文件的映像，它先前的地址空间就会被交换出去。写时复制可以对这种情况进行优化。

fork 和 vfork 的区别：
1. fork()的子进程拷贝父进程的数据段和代码段；vfork()的子进程与父进程共享数据段
2. fork()的父子进程的执行次序不确定；vfork()保证子进程先运行，在调用 exec 或 exit 之前与父进程数据是共享的，在它调用 exec 或 exit 之后父进程才可能被调度运行。
3. vfork()保证子进程先运行，在它调用 exec 或 exit 之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。
4.当需要改变共享数据段中变量的值，则拷贝父进程。

32. 请问如何修改文件最大句柄数？

linux 默认最大文件句柄数是 1024 个，在 linux 服务器文件并发量比较大的情况下，系统会报"too many open files"的错误。故在 linux 服务器高并发调优时，往往需要预先调优 Linux 参数，修改 Linux 最大文件句柄数。
有两种方法：
1. ulimit -n <可以同时打开的文件数>，将当前进程的最大句柄数修改为指定的参数（注：该方法只针对当前进程有效，重新打开一个 shell 或者重新开启一个进程，参数还是之前的值）
首先用 ulimit -a 查询 Linux 相关的参数，如下所示：

```
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 94739
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
```

```
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 94739
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

其中，open files 就是最大文件句柄数，默认是 1024 个。
修改 Linux 最大文件句柄数： ulimit -n 2048， 将最大句柄数修改为 2048 个。

2. 对所有进程都有效的方法，修改 Linux 系统参数
vi /etc/security/limits.conf 添加

```
*    soft    nofile    65536
*    hard    nofile    65536
```

将最大句柄数改为 65536
修改以后保存，注销当前用户，重新登录，修改后的参数就生效了

33. 请问并发(concurrency)和并行(parallelism)

并发（concurrency）：指宏观上看起来两个程序在同时运行，比如说在单核 cpu 上的多任务。但是从微观上看两个程序的指令是交织着运行的，你的指令之间穿插着我的指令，我的指令之间穿插着你的，在单个周期内只运行了一个指令。这种并发并不能提高计算机的性能，只能提高效率。
并行（parallelism）：指严格物理意义上的同时运行，比如多核 cpu，两个程序分别运行在两个核上，两者之间互不影响，单个周期内每个程序都运行了自己的指令，也就是运行了两条指令。这样说来并行的确提高了计算机的效率。所以现在的 cpu 都是往多核方面发展。

34. 请问操作系统是如何进行页表寻址?

页式内存管理，内存分成固定长度的一个个页片。操作系统为每一个进程维护了一个从虚拟地址到物理地址的映射关系的数据结构，叫页表，页表的内容就是该进程的虚拟地址到物理地址的一个映射。页表中的每一项都记录了这个页的基地址。通过页表，由逻辑地址的高位部分先找到逻辑地址对应的页基地址，再由页基地址偏移一定长度就得到最后的物理地址，偏移的长度由逻辑地址的低位部分决定。一般情况下，这个过程都可以由硬件完成，所以效率还是比较高的。页式内存管理的优点就是比较灵活，内存管理以较小的页为单位，方便内存换入换出和扩充地址空间。
Linux 最初的两级页表机制：

两级分页机制将 32 位的虚拟空间分成三段，低十二位表示页内偏移，高 20 分成两段分别表示两级页表的偏移。
* PGD(Page Global Directory): 最高 10 位，全局页目录表索引

* PTE(Page Table Entry)：中间 10 位，页表入口索引

当在进行地址转换时，结合在 CR3 寄存器中存放的页目录(page directory, PGD)的这一页的物理地址，再加上从虚拟地址中抽出高 10 位叫做页目录表项(内核也称这为 pgd)的部分作为偏移，即定位到可以描述该地址的 pgd；从该 pgd 中可以获取可以描述该地址的页表的物理地址，再加上从虚拟地址中抽取中间 10 位作为偏移，即定位到可以描述该地址的 pte；在这个 pte 中即可获取该地址对应的页的物理地址，加上从虚拟地址中抽取的最后 12 位，即形成该页的页内偏移，即可最终完成从虚拟地址到物理地址的转换。从上述过程中，可以看出，对虚拟地址的分级解析过程，实际上就是不断深入页表层次，逐渐定位到最终地址的过程，所以这一过程被叫做 page talbe walk。

Linux 的三级页表机制：

当 X86 引入物理地址扩展(Pisycal Addrress Extension, PAE)后，可以支持大于 4G 的物理内存(36 位)，但虚拟地址依然是 32 位，原先的页表项不适用，它实际多 4 bytes 被扩充到 8 bytes，这意味着，每一页现在能存放的 pte 数目从 1024 变成 512 了(4k/8)。相应地，页表层级发生了变化，Linus 新增加了一个层级，叫做页中间目录(page middle directory, PMD)，变成：

字段	描述	位数
cr3	指向一个 PDPT	crs 寄存器存储
PGD	指向 PDPT 中 4 个项中的一个	位 31~30
PMD	指向页目录中 512 项中的一个	位 29~21
PTE	指向页表中 512 项中的一个	位 20~12
page offset	4KB 页中的偏移	位 11~0

现在就同时存在 2 级页表和 3 级页表，在代码管理上肯定不方便。巧妙的是，Linux 采取了一种抽象方法：所有架构全部使用 3 级页表：即 PGD -> PMD -> PTE。那只使用 2 级页表(如非 PAE 的 X86)怎么办？

办法是针对使用 2 级页表的架构，把 PMD 抽象掉，即虚设一个 PMD 表项。这样在 page table walk 过程中，PGD 本直接指向 PTE 的，现在不了，指向一个虚拟的 PMD，然后再由 PMD 指向 PTE。这种抽象保持了代码结构的统一。

Linux 的四级页表机制：

硬件在发展，3 级页表很快又捉襟见肘了，原因是 64 位 CPU 出现了，比如 X86_64， 它的硬件是实实在在支持 4 级页表的。它支持 48 位的虚拟地址空间 1。如下：

字段	描述	位数
PML4	指向一个 PDPT	位 47~39
PGD	指向 PDPT 中 4 个项中的一个	位 38~30
PMD	指向页目录中 512 项中的一个	位 29~21
PTE	指向页表中 512 项中的一个	位 20~12
page offset	4KB 页中的偏移	位 11~0

Linux 内核针为使用原来的 3 级列表(PGD->PMD->PTE)，做了折衷。即采用一个唯一的，共享的顶级层次，叫 PML4。这个 PML4 没有编码在地址中，这样就能套用原来的 3 级列

表方案了。不过代价就是，由于只有唯一的 PML4, 寻址空间被局限在(239=)512G, 而本来 PML4 段有 9 位, 可以支持 512 个 PML4 表项的。现在为了使用 3 级列表方案，只能限制使用一个， 512G 的空间很快就又不够用了，解决方案呼之欲出。

在 2004 年 10 月，当时的 X86_64 架构代码的维护者 Andi Kleen 提交了一个叫做 4level page tables for Linux 的 PATCH 系列，为 Linux 内核带来了 4 级页表的支持。在他的解决方案中，不出意料地，按照 X86_64 规范，新增了一个 PML4 的层级, 在这种解决方案中，X86_64 拥一个有 512 条目的 PML4, 512 条目的 PGD, 512 条目的 PMD, 512 条目的 PTE。对于仍使用 3 级目录的架构来说，它们依然拥有一个虚拟的 PML4,相关的代码会在编译时被优化掉。这样,就把 Linux 内核的 3 级列表扩充为 4 级列表。这系列 PATCH 工作得不错，不久被纳入 Andrew Morton 的-mm 树接受测试。不出意外的话，它将在 v2.6.11 版本中释出。但是，另一个知名开发者 Nick Piggin 提出了一些看法，他认为 Andi 的 Patch 很不错，不过他认为最好还是把 PGD 作为第一级目录，把新增加的层次放在中间，并给出了他自己的 Patch:alternate 4-level page tables patches。Andi 更想保持自己的 PATCH, 他认为 Nick 不过是玩了改名的游戏，而且他的 PATCH 经过测试很稳定，快被合并到主线了，不宜再折腾。不过 Linus 却表达了对 Nick Piggin 的支持，理由是 Nick 的做法 conceptually least intrusive。毕竟作为 Linux 的扛把子，稳定对于 Linus 来说意义重大。最终，不意外地，最后 Nick Piggin 的 PATCH 在 v2.6.11 版本中被合并入主线。在这种方案中，4 级页表分别是：PGD -> PUD -> PMD -> PTE。

35. 请问线程需要保存哪些上下文，SP、PC、EAX 这些寄存器有什么作用?

线程在切换的过程中需要保存当前线程 Id、线程状态、堆栈、寄存器状态等信息。其中寄存器主要包括 SP PC EAX 等寄存器，其主要功能如下：

SP:堆栈指针，指向当前栈的栈顶地址

PC:程序计数器，存储下一条将要执行的指令

EAX:累加寄存器，用于加法乘法的缺省寄存器

36. 请解释 OS 缺页置换算法

当访问一个内存中不存在的页，并且内存已满，则需要从内存中调出一个页或将数据送至磁盘对换区，替换一个页，这种现象叫做缺页置换。当前操作系统最常采用的缺页置换算法如下：

先进先出(FIFO)算法:置换最先调入内存的页面,即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。

最近最少使用（LRU）算法: 置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。

当前最常采用的就是 LRU 算法。

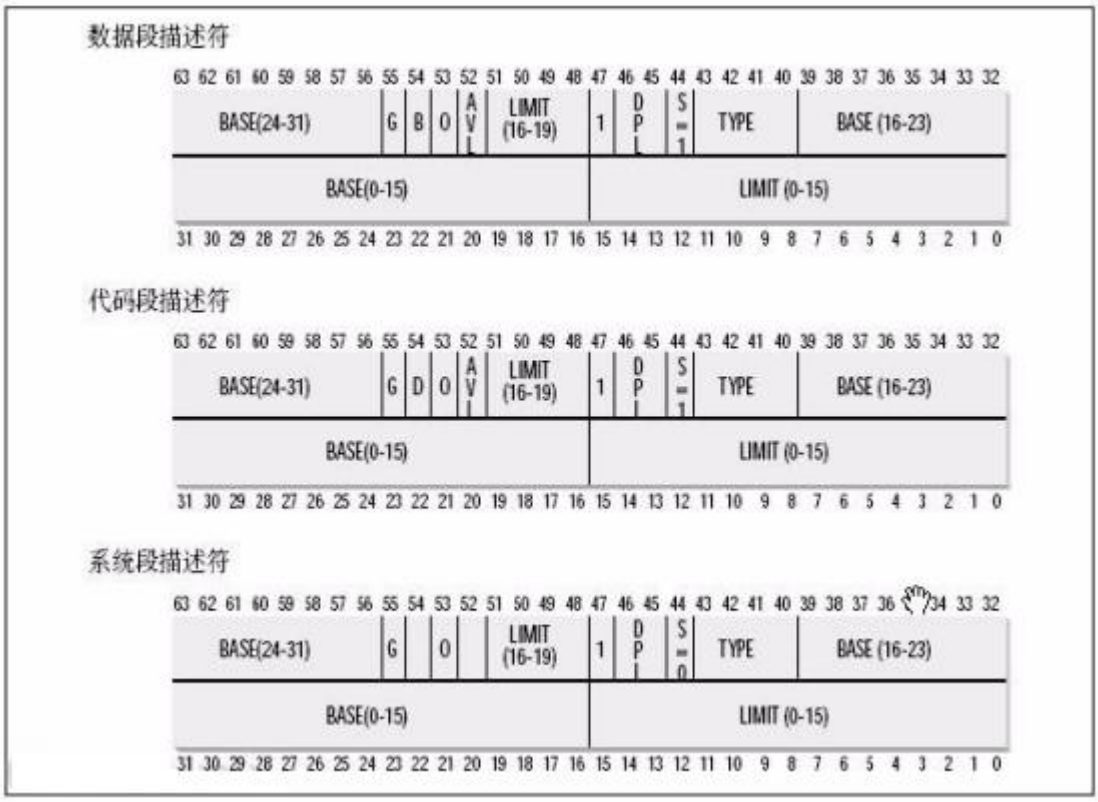
37. 请问虚拟内存和物理内存怎么对应?

1、概念：
物理地址(physical address)
用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。
虽然可以直接把物理地址理解成插在机器上那根内存本身，把内存看成一个从 0 字节一直到最大空量逐字节的编号的大数组，然后把这个数组叫做物理地址，但是事实上，这只是一个硬件提供给软件的抽像，内存的寻址方式并不是这样。所以，说它是“与地址总线相对应”，是更贴切一些，不过抛开对物理内存寻址方式的考虑，直接把物理地址与物理的内存一一对应，也是可以接受的。也许错误的理解更利于形而上的抽像。

虚拟地址(virtual memory)
这是对整个内存（不要与机器上插那条对上号）的抽像描述。它是相对于物理内存来讲的，可以直接理解成“不直实的”，“假的”内存，例如，一个 0x08000000 内存地址，它并不对就物理地址上那个大数组中 0x08000000 - 1 那个地址元素；
之所以是这样，是因为现代操作系统都提供了一种内存管理的抽像，即虚拟内存（virtual memory）。进程使用虚拟内存中的地址，由操作系统协助相关硬件，把它“转换”成真正的物理地址。这个“转换”，是所有问题讨论的关键。
有了这样的抽像，一个程序，就可以使用比真实物理地址大得多的地址空间。甚至多个进程可以使用相同的地址。不奇怪，因为转换后的物理地址并非相同的。
——可以把连接后的程序反编译看一下，发现连接器已经为程序分配了一个地址，例如，要调用某个函数 A，代码不是 call A，而是 call 0x0811111111 ，也就是说，函数 A 的地址已经被定下来了。没有这样的“转换”，没有虚拟地址的概念，这样做是根本行不通的。

2、地址转换
第一步：CPU 段式管理中——逻辑地址转线性地址
CPU 要利用其段式内存管理单元，先将为个逻辑地址转换成一个线程地址。
一个逻辑地址由两部份组成，【段标识符：段内偏移量】。
段标识符是由一个 16 位长的字段组成，称为段选择符。其中前 13 位是一个索引号。后面 3 位包含一些硬件细节，如图：

通过段标识符中的索引号从 GDT 或者 LDT 找到该段的段描述符，段描述符中的 base 字段是段的起始地址
段描述符：Base 字段，它描述了一个段的开始位置的线性地址。
一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中。
GDT 在内存中的地址和大小存放在 CPU 的 gdtr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。
段起始地址+ 段内偏移量 = 线性地址

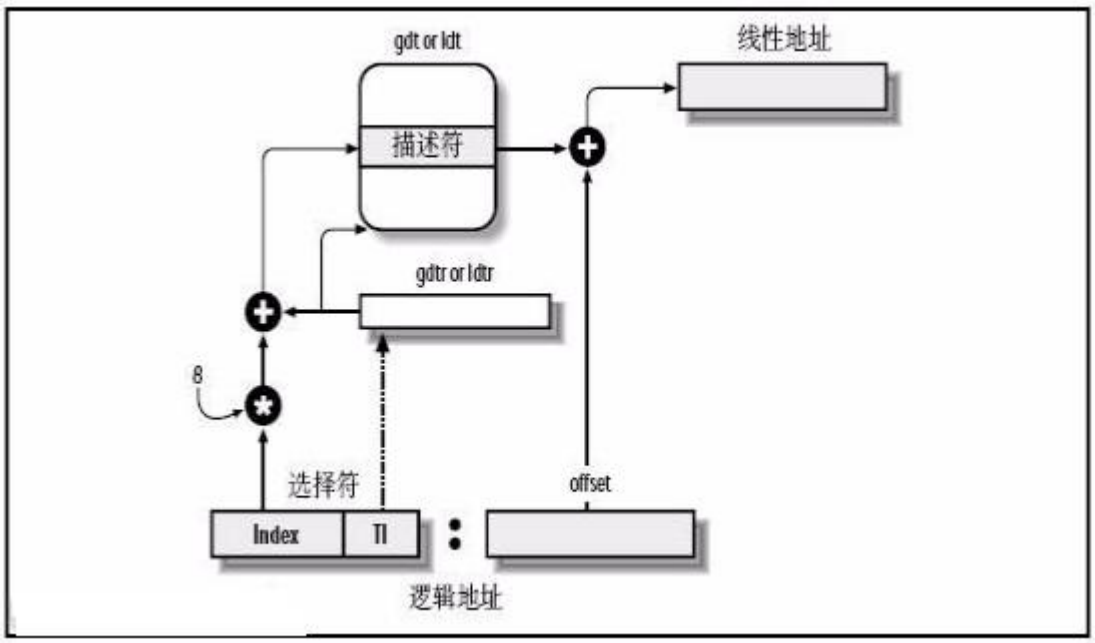


首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]，

1、看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。

2、拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，这样，它了 Base，即基地址就知道了。

3、把 Base + offset，就是要转换的线性地址了。



第一步：页式管理——线性地址转物理地址

再利用其页式内存管理单元，转换为最终物理地址。

linux 假的段式管理

Intel 要求两次转换，这样虽说是兼容了，但是却是很冗余，但是这是 intel 硬件的要求。

其它某些硬件平台，没有二次转换的概念，Linux 也需要提供一个高层抽像，来提供一个统一的界面。

所以，Linux 的段式管理，事实上只是“哄骗”了一下硬件而已。

按照 Intel 的本意，全局的用 GDT，每个进程自己的用 LDT——不过 Linux 则对所有的进程都使用了相同的段来对指令和数据寻址。即用户数据段，用户代码段，对应的，内核中的是内核数据段和内核代码段。

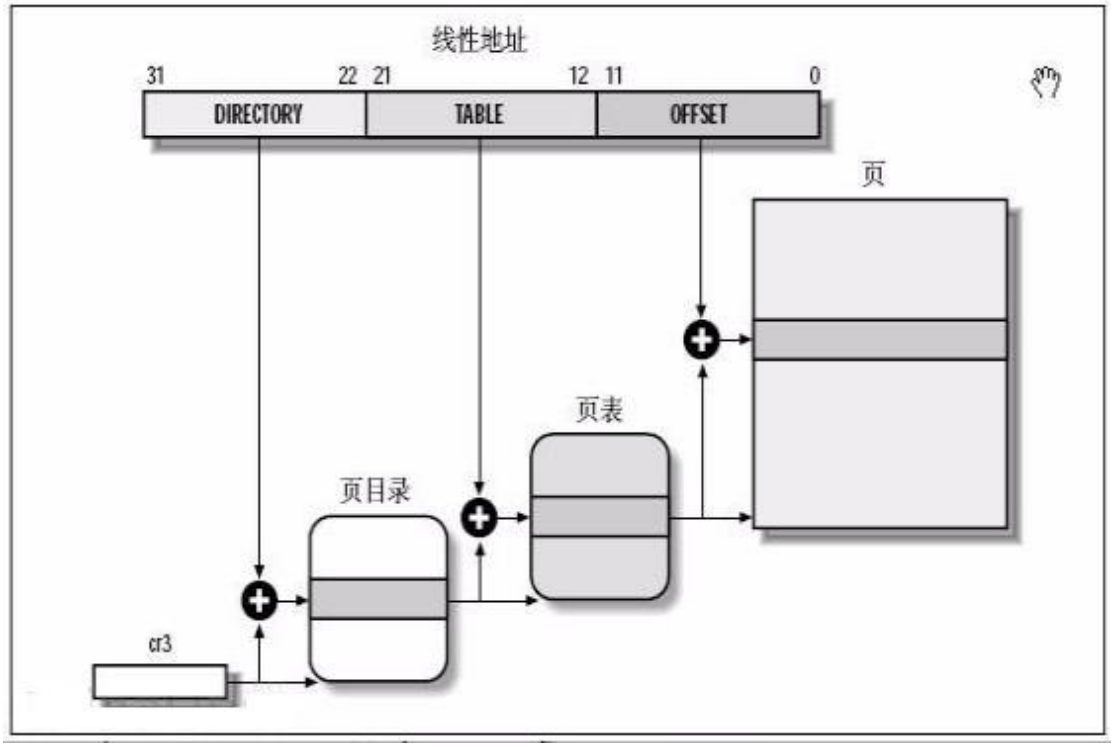
在 Linux 下，逻辑地址与线性地址总是一致的，即逻辑地址的偏移量字段的值与线性地址的值总是相同的。

linux 页式管理

CPU 的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。

线性地址被分为以固定长度为单位的组，称为页(page)，例如一个 32 位的机器，线性地址最大可为 4G，可以用 4KB 为一个页来划分，这页，整个线性地址就被划分为一个 tatol_page[2^20]的大数组，共有 2 的 20 个次方个页。

另一类“页”，我们称之为物理页，或者是页框、页桢的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。



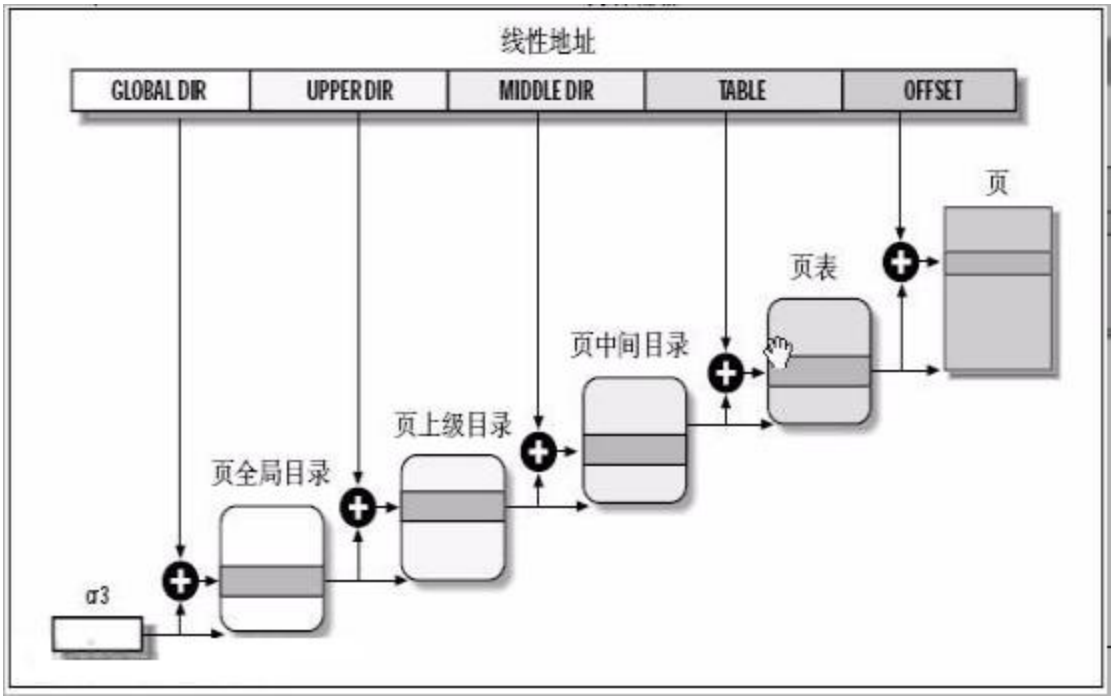
每个进程都有自己的页目录，当进程处于运行态的时候，其页目录地址存放在 cr3 寄存器中。

每一个 32 位的线性地址被划分为三部份，【页目录索引(10 位)：页表索引(10 位)：页内偏移(12 位)】

依据以下步骤进行转换：

从 cr3 中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；

根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。（又引入了一个数组），页的地址被放到页表中去了。
根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；
将页的起始地址与线性地址中最后 12 位相加。
目的：
内存节约：如果一级页表中的一个页表条目为空，那么那所指的二级页表就根本不会存在。这表现出一种巨大的潜在节约，因为对于一个典型的程序，4GB 虚拟地址空间的大部份都会是未分配的；



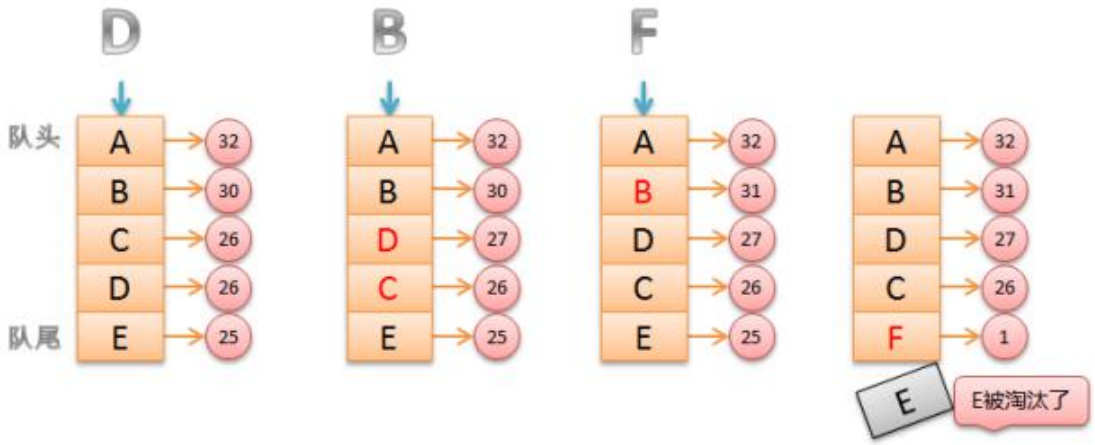
32 位，PGD = 10bit，PUD = PMD = 0，table = 10bit，offset = 12bit
64 位，PUD 和 PMD ≠ 0

38. 请问虚拟内存置换的方式

比较常见的内存替换算法有：FIFO，LRU，LFU，LRU-K，2Q。
1、FIFO（先进先出淘汰算法）
思想：最近刚访问的，将来访问的可能性比较大。
实现：使用一个队列，新加入的页面放入队尾，每次淘汰队首的页面，即最先进入的数据，最先被淘汰。
弊端：无法体现页面冷热信息

2、LFU（最不经常访问淘汰算法）
思想：如果数据过去被访问多次，那么将来被访问的频率也更高。

实现：每个数据块一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序。每次淘汰队尾数据块。
开销：排序开销。
弊端：缓存颠簸。



3、LRU（最近最少使用替换算法）
思想：如果数据最近被访问过，那么将来被访问的几率也更高。
实现：使用一个栈，新页面或者命中的页面则将该页面移动到栈底，每次替换栈顶的缓存页面。
优点：LRU 算法对热点数据命中率是很高的。
缺陷：
1）缓存颠簸，当缓存（1，2，3）满了，之后数据访问（0，3，2，1，0，3，2，1。。。）。
2）缓存污染，突然大量偶发性的数据访问，会让内存中存放大量冷数据。

4、LRU-K（LRU-2、LRU-3）
思想：最久未使用 K 次淘汰算法。
LRU-K 中的 K 代表最近使用的次数，因此 LRU 可以认为是 LRU-1。LRU-K 的主要目的是为了解决 LRU 算法“缓存污染”的问题，其核心思想是将“最近使用过 1 次”的判断标准扩展为“最近使用过 K 次”。
相比 LRU，LRU-K 需要多维护一个队列，用于记录所有缓存数据被访问的历史。只有当数据的访问次数达到 K 次的时候，才将数据放入缓存。当需要淘汰数据时，LRU-K 会淘汰第 K 次访问时间距当前时间最大的数据。
实现：
1）数据第一次被访问，加入到访问历史列表；
2）如果数据在访问历史列表里后没有达到 K 次访问，则按照一定规则（FIFO，LRU）淘汰；
3）当访问历史队列中的数据访问次数达到 K 次后，将数据索引从历史队列删除，将数据移到缓存队列中，并缓存此数据，缓存队列重新按照时间排序；
4）缓存数据队列中被再次访问后，重新排序；
5）需要淘汰数据时，淘汰缓存队列中排在末尾的数据，即：淘汰“倒数第 K 次访问离现在最久”的数据。
针对问题：

LRU-K 的主要目的是为了解决 LRU 算法“缓存污染”的问题，其核心思想是将“最近使用过 1 次”的判断标准扩展为“最近使用过 K 次”。

5、2Q

类似 LRU-2。使用一个 FIFO 队列和一个 LRU 队列。

实现：

1）新访问的数据插入到 FIFO 队列；

2）如果数据在 FIFO 队列中一直没有被再次访问，则最终按照 FIFO 规则淘汰；

3）如果数据在 FIFO 队列中被再次访问，则将数据移到 LRU 队列头部；

4）如果数据在 LRU 队列再次被访问，则将数据移到 LRU 队列头部；

5）LRU 队列淘汰末尾的数据。

针对问题：LRU 的缓存污染

弊端：

当 FIFO 容量为 2 时，访问负载是：ABCABCABC 会退化为 FIFO，用不到 LRU。

39. 编程最长公共连续子序列

```
int substr(string & str1, string &str2)
{
    int len1 = str1.length();
    int len2 = str2.length();
    vector<vector<int>>dp(len1,vector<int>(len2,0));
    for (int i = 0; i < len1; i++)
    {
        dp[i][0] = str1[i]==str1[0]?1:0;
    }
    for (int j = 0; j <= len2; j++)
    {
        dp[0][j] = str1[0]==str2[j]?1:0;
    }
    for (int i = 1; i < len1; i++)
    {
        for (int j = 1; j < len2; j++)
        {
            if (str1[i] == str2[j])
            {
                dp[i][j] = dp[i - 1][j - 1]+1;
            }
        }
    }
    int longest = 0;
    int longest_index = 0;
    for (int i = 0; i < len1; i++)
    {
        for (int j = 0; j < len2; j++)
```

```
            {
                if (longest < dp[i][j])
                {
                    longest = dp[i][j];
                    longest_index = i;
                }
            }
        }

        //字符串为从第 i 个开始往前数 longest 个
        for (int i = longest_index-longest+1; i <=longest_index; i++)
        {
            cout << str1[i] << endl;
        }
        return longest;
    }
```

40. 编程求一个字符串最长回文子串

```
int LongestPalindromicSubstring(string & a)
{
    int len = a.length();
    vector<vector<int>>dp(len, vector<int>(len, 0));
    for (int i = 0; i < len; i++)
    {
        dp[i][i] = 1;
    }
    int max_len = 1;
    int start_index = 0;
    for (int i= len - 2; i >= 0; i--)
    {
        for (int j = i + 1; j < len; j++)
        {
            if (a[i] == a[j])
            {
                if (j - i == 1)
                {
                    dp[i][j] = 2;
                }
                else
                {
                    if (j - i > 1)
                    {
                        dp[i][j] = dp[i + 1][j - 1] + 2;
```

```
                    }
                }
                if (max_len < dp[i][j])
                {
                    max_len =  dp[i][j];
                    start_index = i;
                }
            }
            else
            {
                dp[i][j] = 0;
            }
        }
    }
    cout << "max len is " << max_len << endl;
    cout << "star index is" << start_index << endl;
    return max_len;
}
```

41. 编程如何合并两个有序链表

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(l1 == NULL)
        {
            return l2;
        }
        if(l2 == NULL)
        {
            return l1;
        }
        if(l1->val < l2->val)
        {
            l1->next=mergeTwoLists(l1->next,l2);
            return l1;
        }
        else
        {
            l2->next=mergeTwoLists(l1,l2->next);
            return l2;
        }
    }
};
```

42. 请问什么是单向链表，如何判断两个单向链表是否相交

1、单向链表

单向链表（单链表）是链表的一种，其特点是链表的链接方向是单向的，对链表的访问要通过顺序读取从头部开始；链表是使用指针进行构造的列表；又称为结点列表，因为链表是由一个个结点组装起来的;其中每个结点都有指针成员变量指向列表中的下一个结点。

列表是由结点构成，head 指针指向第一个成为表头结点，而终止于最后一个指向 nuLL 的指针。

2、判断两个链表是否相交

1）方法 1：

链表相交之后，后面的部分节点全部共用，可以用 2 个指针分别从这两个链表头部走到尾部，最后判断尾部指针的地址信息是否一样，若一样则代表链表相交！

2）方法 2：

可以把其中一个链表的所有节点地址信息存到数组中，然后把另一个链表的每一个节点地址信息遍历数组，若相等，则跳出循环，说明链表相交。进一步优化则是进行 hash 排序，建立 hash 表。

43. 有 1 千万条短信，有重复，以文本文件的形式保存，一行一条。请用 5 分钟时间，找出重复出现最多的前 10 条。

```
#include<iostream>
#include<map>
#include<iterator>
#include<stdio.h>
using namespace std;

#define HASH __gnu_cxx
#include<ext/hash_map>
#define uint32_t unsigned int
#define uint64_t unsigned long int
struct StrHash
{
        uint64_t operator()(const std::string& str) const
        {
           uint32_t b    = 378551;
           uint32_t a    = 63689;
           uint64_t hash = 0;
```

```
            for(size_t i = 0; i < str.size(); i++)
            {
               hash = hash * a + str[i];
               a    = a * b;
            }

            return hash;
         }
         uint64_t operator()(const std::string& str, uint32_t field) const
         {
            uint32_t b    = 378551;
            uint32_t a    = 63689;
            uint64_t hash = 0;
            for(size_t i = 0; i < str.size(); i++)
            {
               hash = hash * a + str[i];
               a    = a * b;
            }
                 hash = (hash<<8)+field;
            return hash;
         }
};
struct NameNum{
         string name;
         int num;
         NameNum():num(0),name(""){}
};
int main()
{
         HASH::hash_map< string, int, StrHash > names;
         HASH::hash_map< string, int, StrHash >::iterator it;
         NameNum namenum[10];
         string l = "";
         while(getline(cin, l))
         {
                  it = names.find(l);
                  if(it != names.end())
                  {
                           names[l] ++;
                  }
                  else
                  {
                           names[l] = 1;
```

```
                names[l] = 1;
        }
}
int i = 0;
int max = 1;
int min = 1;
int minpos = 0;
for(it = names.begin(); it != names.end(); ++ it)
{
        if(i < 10)
        {
                namenum[i].name = it->first;
                namenum[i].num = it->second;
                if(it->second > max)
                        max = it->second;
                else if(it->second < min)
                {
                        min = it->second;
                        minpos = i;
                }
        }
        else
        {
                if(it->second > min)
                {
                        namenum[minpos].name = it->first;
                        namenum[minpos].num = it->second;
                        int k = 1;
                        min = namenum[0].num;
                        minpos = 0;
                        while(k < 10)
                        {
                                if(namenum[k].num < min)
                                {
                                        min = namenum[k].num;
                                        minpos = k;
                                }
                                k ++;
                        }
                }
        }
        i++;

}
```

```
        i = 0;
        cout << "maxlength (string,num): " << endl;
        while( i < 10)
        {
                cout << "(" << namenum[i].name.c_str() << ","
                    << namenum[i].num << ")" << endl;
                i++;
        }
        return 0;
}
```

44. 请实现两棵树是否相等的比较，相等返回，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* tree1,TreeNode* tree2);

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

使用递归算法

```
int compTree(TreeNode* tree1,TreeNode* tree2)
{
    if ( tree1==NULL && tree2 == NULL )
        return 0;
    if ( tree1 == NULL || tree2 == NULL )
        return 1;
    if (tree1->c != tree2->c)
        return 1;
    if ( compTree(tree1->leftchild, tree2->leftchild)== 0
        && compTree(tree1->rightchild,tree2->rightchild) == 0 )
        return 0;
    if ( compTree(tree1->leftchild,tree2->rightchild) == 0
        && compTree(tree1->rightchild, tree2->leftchild)== 0 )
        return 0;
}
```

由于需要比较的状态是两棵树的任意状态，而二叉树上的每一个节点的左右子节点都可以交换，因此一共需要对比 2^n 种状态。算法复杂度是 O（2^n）。

45. 时分秒针在一天（24 小时）之内重合多少次？时针与分针又重合了多少次？

```
public void meetTimes() {
    int hTate = 1; // 时针的速度
    int mTate = 12; // 分针的速度
    int sTate = 720; // 秒针的速度
    int circleDis = 720 * 60; // 一圈的路程 = 秒针的速度*60 秒

    int hmsTimes = 0; // 时、分、秒针相遇的次数
    int hmTimes = 0; // 时、分针相遇的次数

    int sDis = 0; // 秒针跳动的总路程
    int mDis = 0; // 分针跳动的总路程
    int hDis = 0; // 时针跳动的总 路程
    float hCurent = 0; // 时针当前所处的刻度位置[0,12)

    for (int i = 1; i <= (24 * 60 * 60); i++) { // 每一次循环跳动一秒
        sDis = sTate * i;
        mDis = mTate * i;
        hDis = hTate * i;

        hCurent = (float) (hDis % circleDis) / (circleDis / 12);

        if ((hDis % circleDis) < (mDis % circleDis)
                && (hDis % circleDis) > ((mTate * (i - 1)) % circleDis))
        { // 时针和分针：划过相遇
            hmTimes++; // 累计次数
            // 打印结果
            printRes("时针和分针第 ", hmTimes, "划过相遇", hCurent);
        } else if ((hDis % circleDis) == (mDis % circleDis)) { // 时针和
            分针：停止相遇
            hmTimes++;
            printRes("时针和分针第 ", hmTimes, "停止相遇", hCurent);

            if ((hDis % circleDis) < (sDis % circleDis)
                    && (hDis % circleDis) > ((sTate * (i - 1)) % circleDis))
            { // 时分秒针：划过相遇
                hmsTimes++;
                printRes("时、分、秒针第 ", hmsTimes, "划过相遇",
                hCurent);
```

```
                } else if ((hDis % circleDis) == (sDis % circleDis)) { // 时
                    分秒针：停止相遇
                    hmsTimes++;
                    printRes("时、分、秒针第 ", hmsTimes, "停止相遇",
                    hCurent);
                }
            }

        }

        System.out.println("一天（24 小时）时、分针相遇的次数 = " + hmTimes);
        System.out.println("一天（24 小时）时、分、秒针相遇的次数 = " +
        hmsTimes);
    }

    /**
     * 打印结果
     */
    public void printRes(String head, int times, String type, float currentLoc)
    {
        DecimalFormat df = new DecimalFormat("0.0000"); // 输出格式
        if (currentLoc == 0) {
            System.out.println(head + times + "次相遇(" + type + ")，相遇位
                    置：0 刻度。");
        } else {
            System.out.println(head + times + "次相遇(" + type + ")，相遇位
                    置："+ df.format(currentLoc) + "刻度。");
        }
    }
```

一天（24 小时）时、分针相遇的次数 = 24
一天（24 小时）时、分、秒针相遇的次数 = 2

时分秒针只可能在 0 点（或 12 点）的位置重合。而时针和分针，除了 0 点和 12 点外，其余每一次重合的位置都不再整点刻度上。

46. 使用 int 做 primary key 和使用 string 有什么优劣？

（1）单实例或者单节点组：

经过 500W、1000W 的单机表测试，自增 ID 相对 UUID 来说，自增 ID 主键性能高于 UUID，磁盘存储费用比 UUID 节省一半的钱。所以在单实例上或者单节点组上，使用自增 ID 作为首选主键。

（2）分布式架构场景：

20 个节点组下的小型规模的分布式场景，为了快速实现部署，可以采用多花存储费用、牺牲部分性能而使用 UUID 主键快速部署；
20 到 200 个节点组的中等规模的分布式场景，可以采用自增 ID+步长的较快速方案。
200 以上节点组的大数据下的分布式场景，可以借鉴类似 twitter 雪花算法构造的全局自增 ID 作为主键。

47. 如何预估一个 mysql 语句的性能？

Explain 命令在解决数据库性能上是第一推荐使用命令，大部分的性能问题可以通过此命令来简单的解决，Explain 可以用来查看 SQL 语句的执行效 果，可以帮助选择更好的索引和优化查询语句，写出更好的优化语句。
Explain 语法：explain select … from … [where ...]
例如：explain select * from news;
输出：

```
+----+-------------+-------+-------+-----------------+---------+---------
+-------+------+-------+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows
| Extra |
+----+-------------+-------+-------+-----------------+---------+---------
+-------+------+-------+
```

下面对各个属性进行了解：
1、id：这是 SELECT 的查询序列号
2、select_type：select_type 就是 select 的类型，可以有以下几种：
SIMPLE：简单 SELECT(不使用 UNION 或子查询等)
PRIMARY：最外面的 SELECT
UNION：UNION 中的第二个或后面的 SELECT 语句
DEPENDENT UNION：UNION 中的第二个或后面的 SELECT 语句，取决于外面的查询
UNION RESULT：UNION 的结果。
SUBQUERY：子查询中的第一个 SELECT
DEPENDENT SUBQUERY：子查询中的第一个 SELECT，取决于外面的查询
DERIVED：导出表的 SELECT(FROM 子句的子查询)

3、table：显示这一行的数据是关于哪张表的
4、type：这列最重要，显示了连接使用了哪种类别,有无使用索引，是使用 Explain 命令分析性能瓶颈的关键项之一。
结果值从好到坏依次是：
system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL
一般来说，得保证查询至少达到 range 级别，最好能达到 ref，否则就可能会出现性能问题。
5、possible_keys：列指出 MySQL 能使用哪个索引在该表中找到行
6、key：显示 MySQL 实际决定使用的键（索引）。如果没有选择索引，键是 NULL

7、key_len：显示 MySQL 决定使用的键长度。如果键是 NULL，则长度为 NULL。使用的索引的长度。在不损失精确性的情况下，长度越短越好
8、ref：显示使用哪个列或常数与 key 一起从表中选择行。
9、rows：显示 MySQL 认为它执行查询时必须检查的行数。
10、Extra：包含 MySQL 解决查询的详细信息，也是关键参考项之一。
Distinct
一旦 MYSQL 找到了与行相联合匹配的行，就不再搜索了
Not exists
MYSQL 优化了 LEFT JOIN，一旦它找到了匹配 LEFT JOIN 标准的行，
就不再搜索了
Range checked for each
Record（index map:#）
没有找到理想的索引，因此对于从前面表中来的每一 个行组合，MYSQL 检查使用哪个索引，并用它来从表中返回行。这是使用索引的最慢的连接之一
Using filesort
看 到这个的时候，查询就需要优化了。MYSQL 需要进行额外的步骤来发现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来 排序全部行
Using index
列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表的全部的请求列都是同一个索引的部分的时候
Using temporary
看到这个的时候，查询需要优化了。这 里，MYSQL 需要创建一个临时表来存储结果，这通常发生在对不同的列集进行 ORDER BY 上，而不是 GROUP BY 上
Using where
使用了 WHERE 从句来限制哪些行将与下一张表匹配或者是返回给用户。如果不想返回表中的全部行，并且连接类型 ALL 或 index， 这就会发生，或者是查询有问题

48. 一个环，有 n 个点，每次只能走一步， 问从原点 0 出发，经过 k 步回到原点有多少种方法？

现在把环上的点编号为 0 到 n-1, 即从 0 点出发，再回到 0 点有多少种方法？
我们可以想到，再回到 0 点可以从右面回来，也可以从左面回来，即先到达旁边的一个点，看看有多少回来的方法即可。所以运用动态规划的思想，我们可以写出递推式如下：
d(k, j) = d(k-1, j-1) + d(k-1, j+1);
d(k, j)表示从点 j 走 k 步到达原点 0 的方法数， 因此可以转化为他相邻的点经过 k-1 步回到原点的问题，这样将问题的规模缩小。由于是环的问题， j-1, j+1 可能会超出 0 到 n-1 的范围，因此，我们将递推式改成如下：
d(k, j) = d(k-1, (j-1+n)%n) + d(k-1, (j+1)%n);
因为问题从走 k 步转化为走 k-1 步的问题，所以在写程序的时候我们就按照 k 从 0 开始递增的循环写，这样当计算第 k 步的时候可以直接使用 k-1 步的结果。
实例如下， n = 3, k = 4

按照 k 增大计算

	0	1	2
k=0	1	0	0
k=1	0	1	1
k=2	2	1	1
k=3	2	3	3
k=4	6	5	5

因为计算第 k 步只与第 k-1 步的值有关，因此可以使用两行的数组来存储。代码如下：

```
/**
 * 一个圆环， 有 n 个点， 从 0 出发，每次只能走一步，问走 k 步，有多少种方法可以走回来
 */
#define N 100

int get_step_num(int n, int k)
{
    if (n==1){
        return 1;
    }

    if (n==2){
        if (k%2==0)
            return 1;
        else
            return 0;
    }

    int arr[2][N] = {0};
    int flag = 1, i = 0, j = 0;

    arr[0][0] = 1;
    for (i=1; i<n; i++){
        arr[0][i] = 0;
    }

    // j is the current step
    for (j=1; j<=k; j++){
        for (i=0; i<n; i++){
            arr[flag][i] = arr[!flag][(i-1+n)%n] + arr[!flag][(i+1)%n];
        }
        flag = !flag;
    }
```

```
    return arr[!flag][0];
}

int main()
{
    int n, k;
    printf("Please input the number n and the step k:\n");
    scanf("%d%d", &n, &k);
    printf("%d\n", get_step_num(n, k));
    return 0;
}
```

49. 给你一个有序整数数组，数组中的数可以是正数、负数、零，请实现一个函数，这个函数返回一个整数：返回这个数组所有数的平方值中有多少种不同的取值。

举例：
nums = {-1,1,1,1},
那么你应该返回的是：1。因为这个数组所有数的平方取值都是 1，只有一种取值
nums = {-1,0,1,2,3}
你应该返回 4，因为 nums 数组所有元素的平方值一共 4 种取值：1,0,4,9

方法一：暴力法，先算平方和，保存在一个数组中，然后使用集合统计不同。

方法二：集合保存平方和。直接先遍历一次得到平方和，将平方和放入集合中，输出集合的大小。时间复杂度为 O(n),空间复杂度为 O(n).

方法三：以上两种方法都没有使用到数组有序，需要计算的是不同的平方和，那么平方和相同时有两种情况，一：两个元素是相邻元素，元素值本身相同；二：两个元素绝对值相等；对于第一种情况可以直接比较是否与相邻元素相等，相等时不计数；对于第二种情况利用数组有序的条件，使用双指针分别指向数组的头和尾，相等时不计数。两指针向中间移动。

```
public static int DifferentMi(int nums[]){
    int cnt=0;
    int i=0,j=nums.length-1;
    while(i<j){
        while(i<j && nums[i]*nums[i]==nums[j]*nums[j])
            i++;
        if(nums[i]*nums[i]>nums[j]*nums[j]){
            while(i<j && nums[i]*nums[i]==nums[i+1]*nums[i+1])
                i++;
```

```
                i++;
            }else{
                while(i<j && nums[j]*nums[j]==nums[j-1]*nums[j-1])
                    j--;
                j--;
            }
            cnt++;
        }
        return cnt;
    }
```

50. 现在有一个数组，已知一个数出现的次数超过了一半，请用 O(n)的复杂度的算法找出这个数。

分析：设数 A 出现次数超过一半。每次删除两个不同的数，在剩余的数中，数 A 出现的次数仍超过一半。通过重复这个过程，求出最后的结果。这个题目与编程之美中寻找水王相同

乘以 2.

```
#include<iostream>
using namespace std;
//size 为数组 A 的大小
//返回数组中出现超过一半的数
int search(int *A,int size)
{
    int count=0;
    int current;
    for(int i=0;i<size;i++)
    {
        if(count==0)
        {
            current=A[i];
            count=1;

        }
        else
        {
            if(A[i]==current)
                count++;
            else
                count--;
        }
```

```
    }
    return current;
}
int main()
{
    int A[6]={1,2,2,1,1,1};
    int B[7]={1,0,1,0,0,1,1};
    int C[7]={3,4,6,3,3,3,7};
    cout<<search(A,6)<<" ";
    cout<<search(B,7)<<" ";
    cout<<search(C,7)<<" "<<endl;
    int i;
    cin>>i;
    return 0;
}
```

获取更多资料，请联系【零声学院】Milo 老师 QQ:472251823