

知识点回顾

1. 数据结构

@面向对象的数据结构一般设计思路:

基本数据成员 (数据属性/成员/域)

数据存储:

线性结构 (数组、链表、栈、队列)

树状结构 (二叉树、堆)

无序结构 (集合、哈希表)

数据操作/函数/方法 (增、删、查、改)

面向对象术语对: 属性/操作 (theory)、成员/函数 (C++)、域/方法 (Java)

@数据结构分类

“非关联结构”

线性结构 数组 栈 队列

树状结构 二叉树 B 树 堆 并查集 Trie 树

网状结构 图

“无序结构” 集合 哈希表

关联结构 键值对 key-value pair

映射 Map HashMap/ TreeMap

字典?

红黑树专题

5 大性质:

左旋、右旋

插入

删除

与 AVL 树比较

应用:

栈 平衡符号[()], 后缀表达式 (逆波兰表达式)、中缀到后缀的转换、方法调用

队列 排队论 (queuing theory)

堆 堆数据结构优先权队列、选择算法(最值)、prim 最小生成树、dijkstra、排序算法中有重要应用。

Java 对 heap 的实现是 PriorityQueue 类, 阅读其源码。

创建堆的时间复杂度为 $O(N)$, 《数据结构与算法分析——Java 语言描述》定理 6.1

堆的上滤 (precolate up) 上滤 d 层, 若交换, 赋值次数 $3d$, 若“移位”, 赋值次数 $d+1$

优先权队列如何实现? 插入元素, 末尾, 自底向上调整; 删除元素, 交换首尾元素, 自顶向下调整。

树、森林定义是什么? 如何存储表示?

判断一个图是否一棵树

无重边、无环、仅根节点入度为 0、非森林
树的典型应用就是文件系统

二叉树、AVL 树、B 树、B+树、红黑树如何实现？二叉查找树的用途？（删除操作比较复杂）二叉搜索树（Binary Search Tree）或二叉排序树（Binary Sorted Tree）
线索二叉树？Huffman 树？它们的用途？

已知二叉树的先序遍历，求其可能的中序遍历和可能的后序遍历。

二叉查找树 删除节点 2 个思路：用左子树的最大节点或者右子树的最小节点代替待删除节点。一般使用后者，因为右子树中的最小节点不可能有左儿子，更容易处理。

二叉查找树的所有节点的期望深度为 $O(\log N)$ 。（等概率，递归分析）

平衡树 AVL 树的高度与最少节点数之间的递推关系，与斐波那契数列密切相关

并查集

应用：求无向图的连通分量个数、[最近公共祖先](#) (Least Common Ancestor, Robert Tarjan) (POJ 1330)、带限制的作业排序、实现 Kruskal 算法求最小生成树等。
<http://dongxicheng.org/structure/union-find-set/>

哈希 任意长度的输入 → 固定长度的输出

任意长度的消息 → 固定长度的消息摘要

关键字 $k \rightarrow f(k)$ 地址/存储位置

复习 HashMap:

`key > key.hashCode > hash()`

`> indexOf(hash, table.length)`

key 映射到 HashMap 的内部数组的下标，每个数组元素是一个链表节点，相同关键字保存在同一个桶中。拉链法解决散列表冲突。

散列函数的设计

horner 法则 进制转换

解决冲突

再散列、可扩散列（实际）

实用散列技术

开放散列：分离链接+再散列

封闭散列：开放地址（平方探测、双散列）+再散列

再散列把程序员从对表大小的担心中解放出来。

再散列的时机

散列表 编译器(符号表)、节点有实名而非数字的图论问题、游戏编制(变换表)、在线拼写检查。第五章 散列 5.13 散列模式串 5.18 拼写检查

哈希就是将一个记录的键值映射到其存储位置的过程。

哈希的 “No”：键值相同的多个记录，范围查找，最值查找。

哈希的 “Yes”：精确查找，内存或磁盘，数据库（哈希，B 树）。

@如何判断图的连通性 并查集 深度优先搜索
@三数中值分割法 排序 选择
@希尔排序, Hibbard 增量序列, Sedgewick 增量序列
最坏情形时间复杂度
堆垒数论 (additive number theory)
@二叉查找树可以迅速找到一定范围内的所有项?
中缀表达式 栈 二叉树
懒惰删除策略
@食物链
数据如何存储并高效检索
@离散数学 关系 向量 并查集
@赢者树与堆的性能比较
赢者树创建复杂度 $O(k)$, 建堆 $O(n)$
赢者树不需要节点交换, 比堆的性能要好
赢者树能够求第 k 个最值, 堆却不能。
赢者树这块, 你还不熟练
输出 $1b$ 个数据的前 k -th 数据
Java 有已经实现的赢者树吗?

2. 算法

算法分类:
递归与分治
回溯
分支限界
贪心
动态规划
KMP

最近公共祖先

@n Queen k Knight 问题

@Queen 问题和 Knight 问题

深度优先、广度优先

复杂度分析

@NP 代表 nondeterministic polynomial-time 非确定型多项式时间的

NP 完全问题

哈密尔顿回路问题、巡回售货员问题、最长路径问题、装箱问题 (bin packing)、

背包问题 (knapsack)、图的着色 (graph coloring)、

团问题 (clique) ?

@ 0-1 背包问题 (先写递归, 分析最优子结构和重叠子问题, 再写动态规划)

1. 选择问题 (selection problem)

k 次冒泡排序

先把前 k 个元素读入一个数组, 再逐个处理剩下的元素

@第 k 大元素 Knuth 卷 3 5.3.3

2. 最大的子序列和问题

给定整数 A_1, A_2, \dots, A_n (可能有负数), 求 $\sum_{k=i}^j A_k$ 的最大值 (为方便起见, 如果所有整数均为负数, 则最大子序列和为 0)。

例如:

对于输入 -2, 11, -4, 13, -5, -2, 答案为 20 (从 $A_2 \sim A_4$)。

[最大子序列和程序](#)

@输出前 n 个质数、fibonacci 数

@《算法导论》最长上升子序列

@全排列 整数分解 最大公约数

@LCA RMQ 问题

1. 二分查找

2. 欧几里得算法 -> 密码学

3. [求素数](#) 1) 素数生成法; 2) 埃拉托斯特尼(Eratosthenes)《编程珠玑》

4. [幂运算](#) (递归+分治)

@[Kruskal 算法](#), 用到并查集和堆。

深度优先生成树

@字符串匹配比较经典的算法有: Knuth-Morris-Pratt 算法, Boyer-Moore 算法, Karp-Rabin 算法。其中 Karp-Rabin 算法是用散列, 那个位图算法真没见过, 谁来讲解一下?

@复习 KMP 算法

字符串匹配 KMP 算法 String 类又是如何实现的?

串搜索算法 维基百科

有限自动机实现?

还有其他很多算法

String 类的解读, 看博客 KMP 算法

@数组去重 基数估计 位图

排序

归并排序、冒泡排序、插入排序、基数排序 (稳定)

快速排序、堆排序、选择排序、希尔排序 (不稳定)

@原地归并 块交换 近年考研题

@随机排列, 如何实现?

@自然排序 [picture1.jpg](#) [picture2.jpg](#) [picture11.jpg](#)

@比较 文件系统 数据库系统

存储海量 QQ 号码的处理性能

例如外排序

MySQL 实现外排序

hadoop 实现外排序

@快速排序，两路归并排序，堆排序(yes)

@外部排序

复习希尔排序

3. 编译、连接和加载

@由于对象变量本身在栈上分配内存，所以在销毁对象时按照创建时的逆序进行，即最晚创建的对象最早销毁。

@多态举例，各种图形实现同一个接口或者继承同一个抽象类，用一个 list 加到窗口中，调用它们的 draw 方法。

@C 语言中未初始化的指针，指向 NULL 吗？探究！C 语言的源码？有没有研究这个问题。

@动态连接与静态连接的比较

@构造方法为什么不能是 static

@尾递归：函数体中最后执行的语句是仅包含自身调用。

举例：

```
long tailRecursive(long n, long a) {  
    if (n < 0) return 0;  
    else if (n == 0) return 1;  
    else if (n == 1) return a;  
    else return tailRecursive(n - 1, a * n);  
}  
tailRecursive(5, 1);
```

@对指针和数组变量进行+1 操作，分别产生什么变化？

4. 操作系统

进程

进程的状态：就绪、运行、阻塞

进程状态机

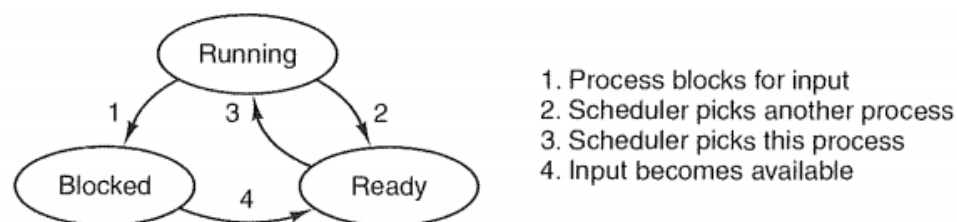


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

进程和线程的区别：

- 1) 进程是线程的容器，程序只有创建了进程之后才能创建线程；
- 2) 进程是操作系统资源（内存、IO）分配的单位，线程是操作系统 CPU 调度的单位；
- 3) 进程间通信的手段有管道、消息队列、共享内存、套接字、信号量，同一个进程中的线程间通信：共享全局变量？互斥量、信号量、条件变量？

参考：<http://blog.csdn.net/wonderwander6642/article/details/8008241>

死锁

产生死锁的原因：产生死锁的原因：1) 竞争资源；2) 进程间推进顺序非法。

产生死锁的必要条件：互斥条件、保持和等待条件、非剥夺条件、循环等待。

处理死锁的方法：预防死锁、避免死锁（银行家算法）、检测死锁、解除死锁

死锁举例：A(b)->B(c)->C(a)

同步与互斥

5. 数据库

@复习 SQL 语句

执行顺序：

from where group by having select order by

@两个表的迪卡尔积是怎么回事？表是行（元组）的集合

@尝试用 MongoDB 作为微博后台数据库

@如何设计一个微博数据库

@如何分析数据库的查询效率？

@数据库中的存储过程、索引分别是怎么回事？join、union 关键字？

@有这么一道题 - 微博数据库设计：有 A,B,C3 个用户，A 关注 C，C 关注 A 和 B；A,B 更新后 C 会收到信息提示，比如：

2010-11-16 22:40 用户 A 发表 a1;

2010-11-16 22:41 用户 A 发表 a2;

2010-11-16 22:42 用户 A 发表 a3

2010-11-16 23:40 用户 B 发表 b1;

2010-11-16 22:40 用户 B 发表 b2;

问题 1：如何设计数据表和查询？

问题 2：如果 C 关注了 10000 个用户，A 被 10 万个人关注，系统又该如何设计？

答 1：

user: user_id, others

microblog: microblog_id, date_time, user_id, content, others

following: follower_id, followee_id, date_time, others

谈谈你对范式的理解：

查询：

```
SELECT * FROM microblog WHERE user_id = current_user_id OR user_id IN (SELECT followee_id WHERE follower_id = current_user_id) ORDER BY data_time DESC LIMIT start_index, page_size
```

如果用表的连接，查询语句又该如何写？

6. 软件工程/面向对象分析/UML 建模

1. UML 类图中，常见的关系

泛化（Generalization）实线+空心三角箭头（指向父类） 例子：老虎是动物的一种

实现（Realization）虚线+空心三角箭头（指向接口） 例子：

关联（Association）实线（双向）或实线+箭头（单向） 例子：老师和学生、丈夫与妻子

聚合（Aggregation）实线+空心菱形箭头（指向整体） 例子：车和轮胎

组合（Composition）实线+实心菱形箭头（指向整体） 例子：公司和部门

依赖（Dependency）虚线+箭头（指向依赖对象） 例子：现代人依赖计算机

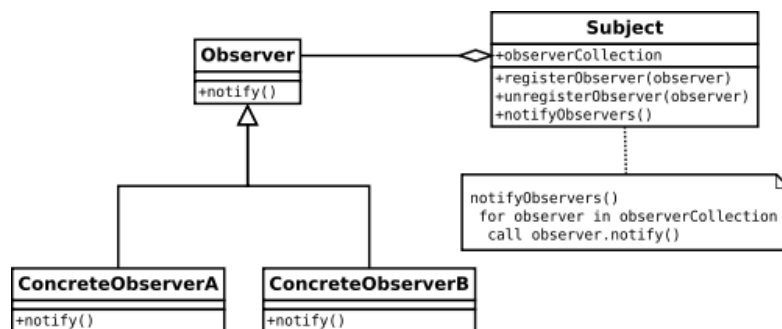
7. 设计模式

观察者模式

Java 中已有的类：Observable（事件源）、Observer（观察者接口）

1）事件源把每一个观察者的引用添加到一个观察者集合中；

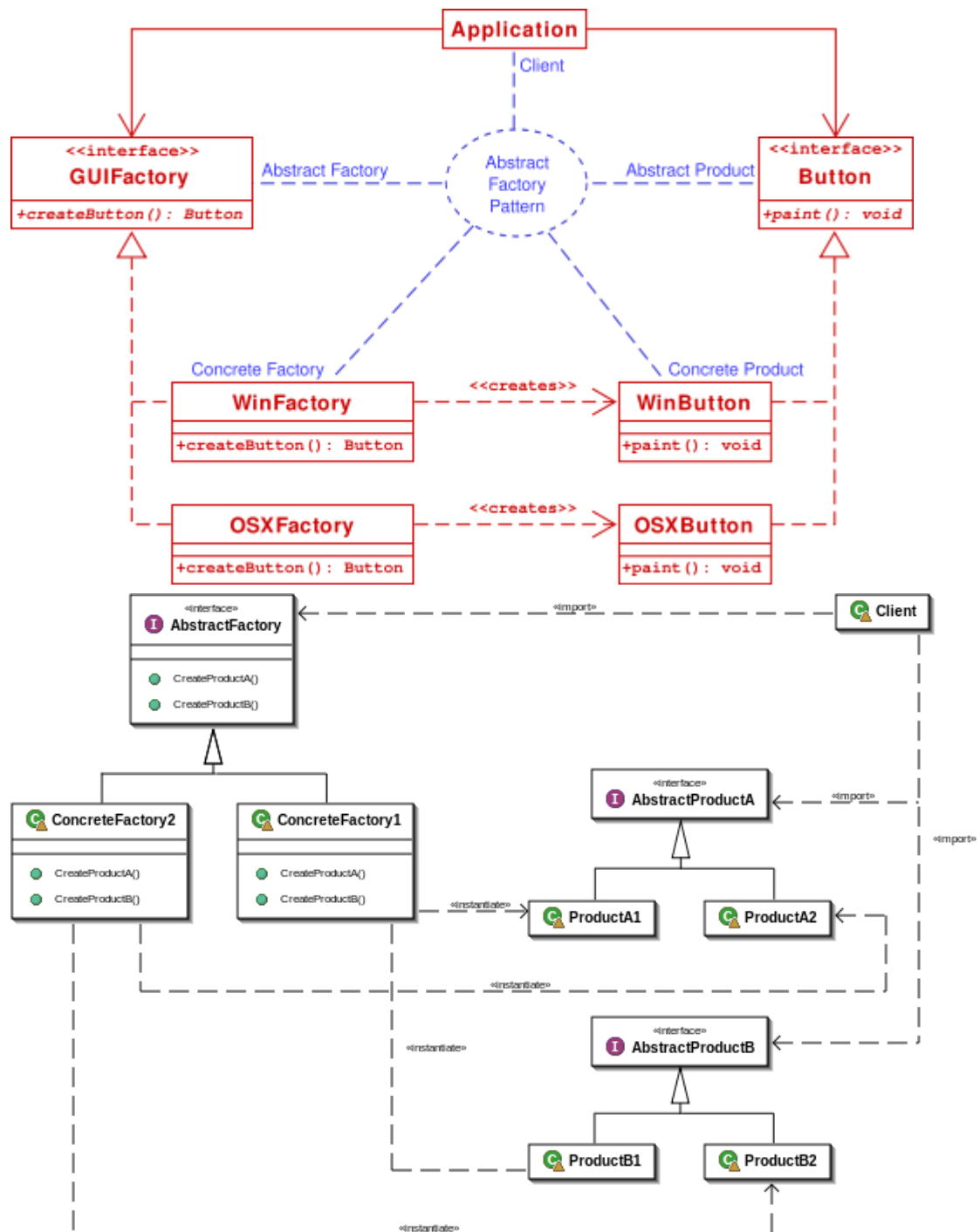
2）当产生一个新的事件时，事件源就会调用观察者集合中每一个观察者的通知方法。



接口回调模式

A -> B b(A a) { a.callback();}

抽象工厂



@控制反转: 依赖查找(回调)和依赖注入(setter injection, constructor injection)。

XML>工厂模式>依赖对象

@访问磁盘文件、网络资源、查询数据库都是影响应用程序执行性能的重要因素。

文件 IO、数据库 IO、网络 IO

@事件驱动机制(被动)

主动 3 种

Windows

message-based event-driven

Java 监听机制 观察者模式

Qt signal/slot

@应用程序框架 application framework

X Window: Qt motif openwin gtk

Windows: mfc owl vcl atl

Java: swing

@查看 Button 类如何实现的 尝试实现自己的

@observer pattern

Observable 类（事件源）Observer 接口（观察者）

@Button 按钮，谁是观察者，谁是被观察者？按钮是被观察者。

婴儿哭，爸爸、妈妈、狗的反应。

@Java Swing MVC 框架模式？不是一个设计模式？

8. 计算机网络

当输入一个网址，实际会发生什么？

1) 通过域名得到 IP 地址，DNS 查找过程：浏览器缓存、系统缓存、路由器缓存、ISP DNS 缓存、递归搜索（根域名服务器、.org 域名服务器、wikipedia.org 域名服务器）

一个域名可能对应多个 IP 地址，即采用 CDN 技术来提高负载均衡。

2) http 请求 tcp 数据段 ip 数据包 以太网数据帧

协议数据单元 PDU（Protocol Data Unit）是指对等层次之间传递的数据单位。协议数据单元(Protocol Data Unit)物理层的 PDU 是数据位(bit)，数据链路层的 PDU 是数据帧(frame)，网络层的 PDU 是数据包(packet)，传输层的 PDU 是数据段(segment)，其他更高层次的 PDU 是数据(data)。

图 36.1. TCP/IP 协议栈

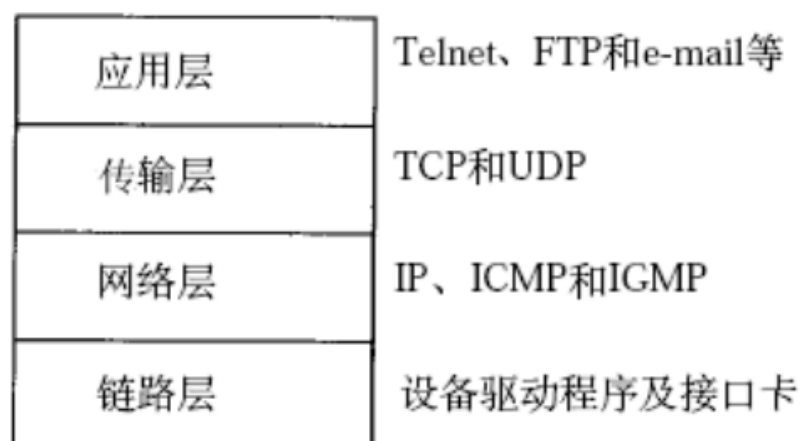


图 36.2. TCP/IP 通讯过程

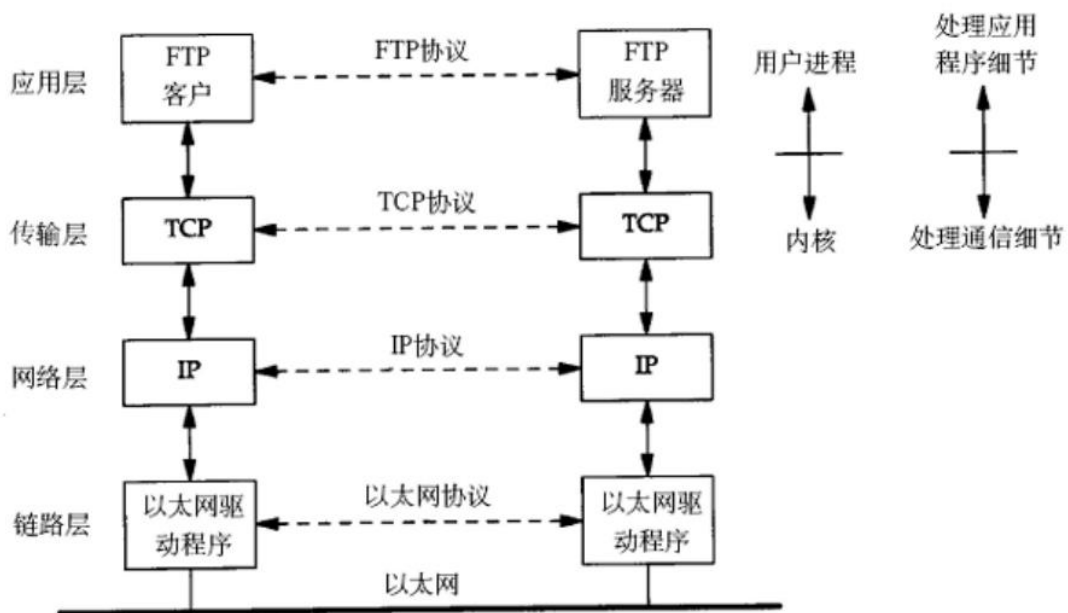
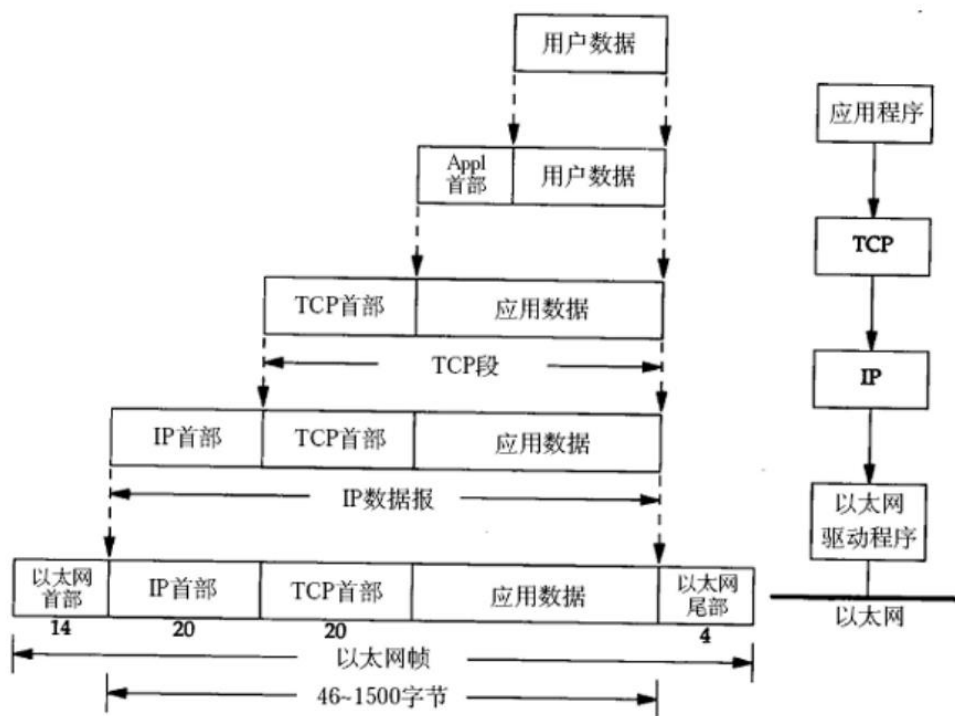
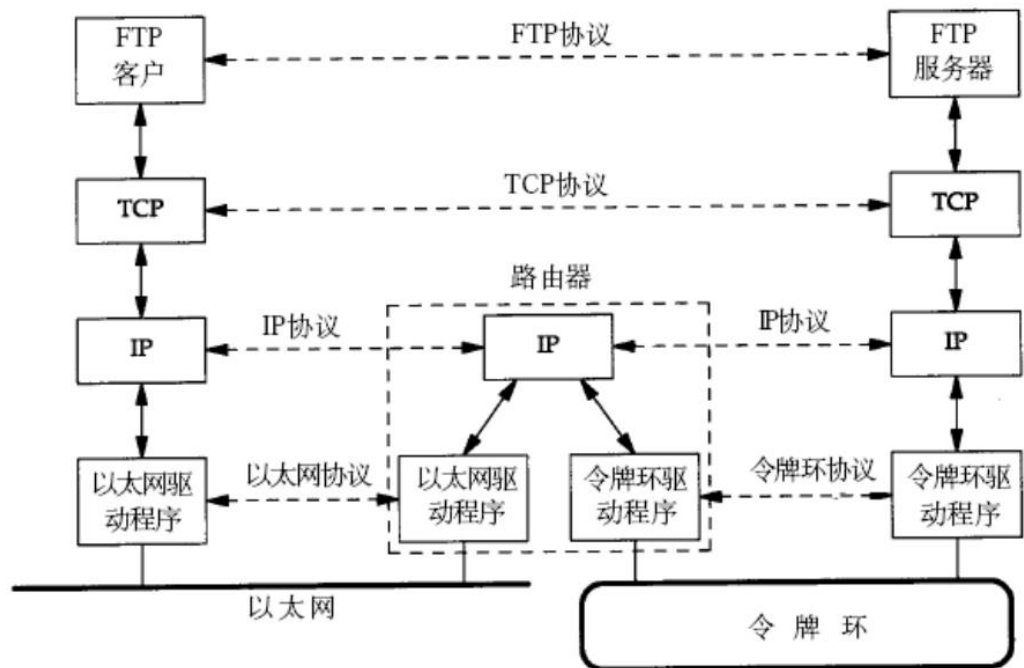


图 36.3. TCP/IP 数据包的封装



跨路由器通信过程



路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包，因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。

网络层负责点到点（point-to-point）的传输（这里的“点”指主机或路由器），而传输层负责端到端（end-to-end）的传输（这里的“端”指源主机和目的主机）。

图 36.6. 以太网帧格式

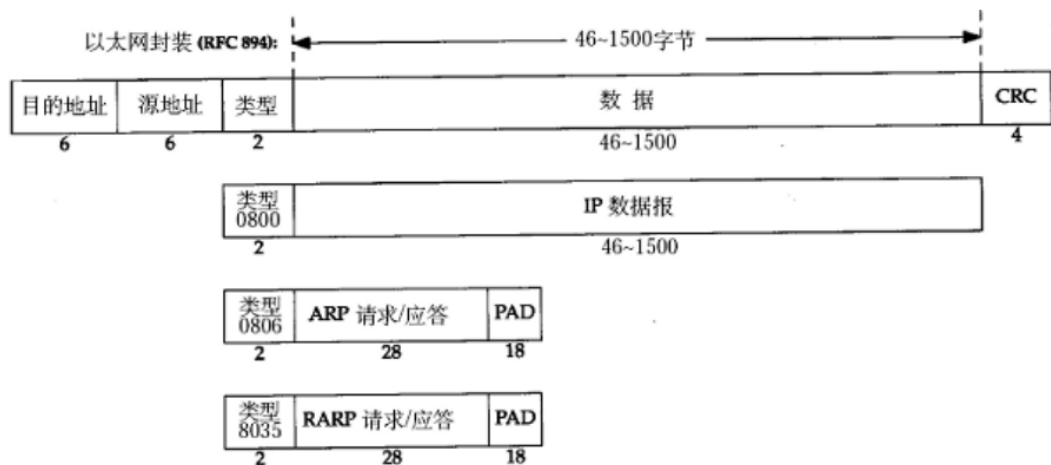
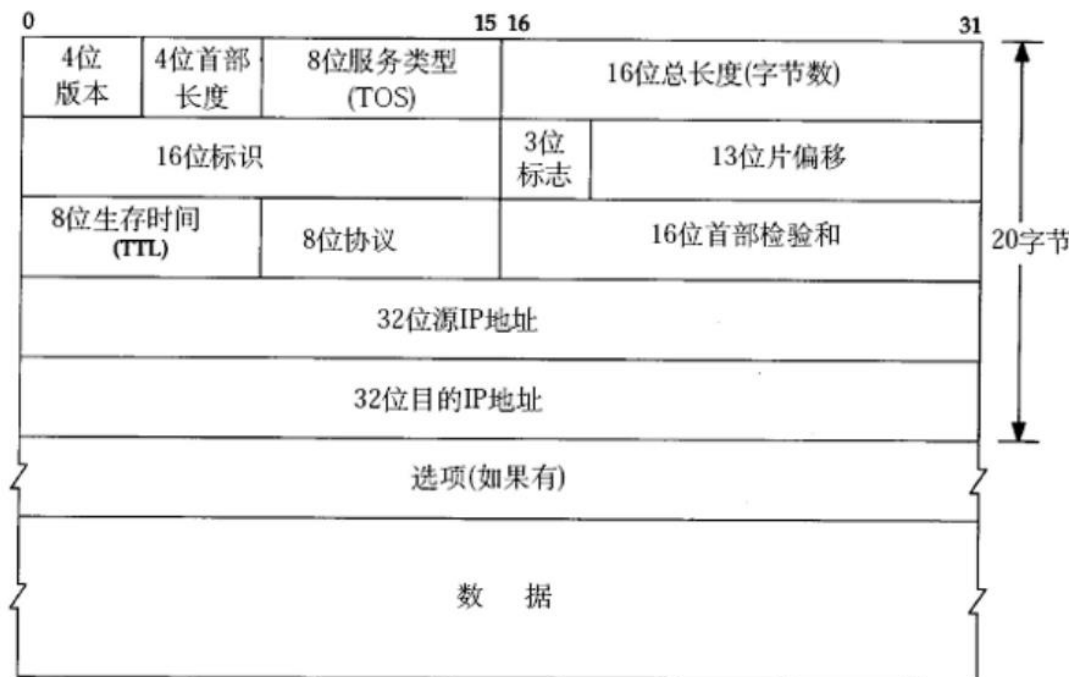


图 36.8. IP 数据报格式



IP 数据报的总长度用 2 个字节表示，意味着 IP 数据报最长为 65535 个字节。

图 36.11. UDP 段格式

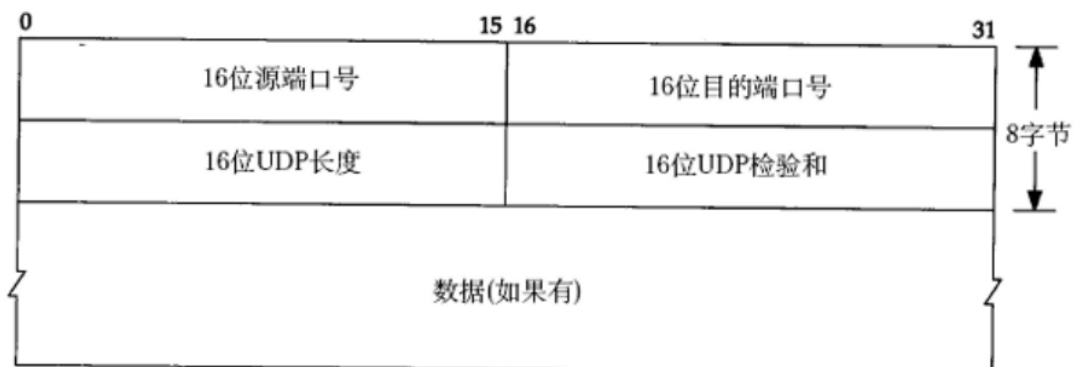
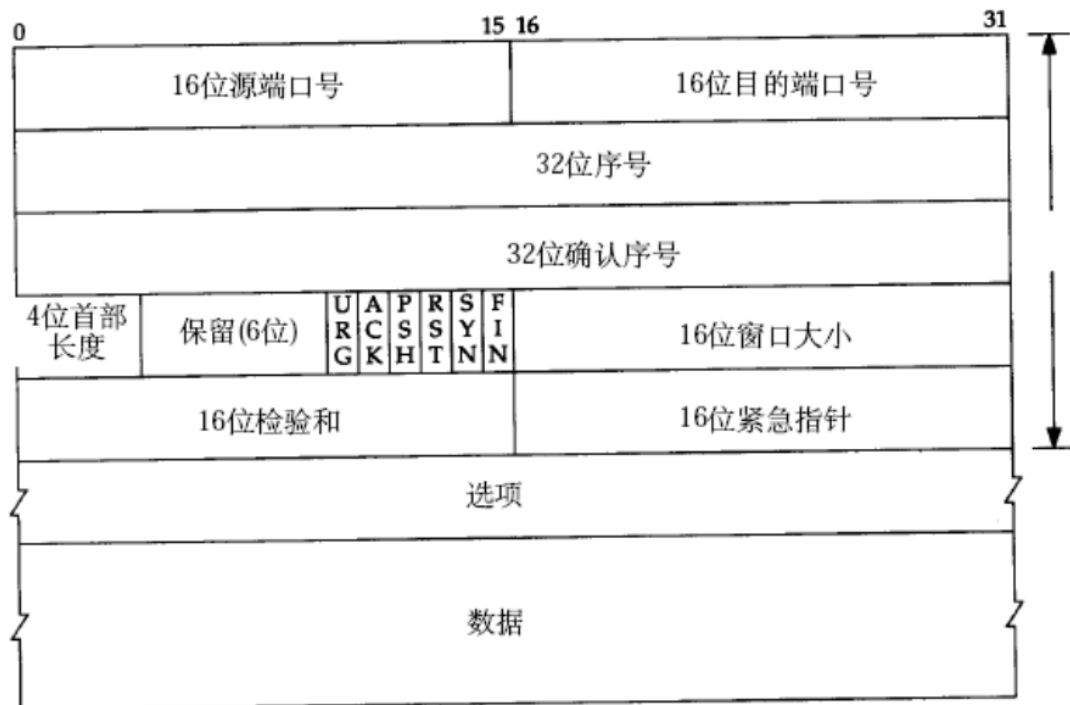


图 36.12. TCP 段格式



9. Java 专题

Java 与 C++的区别

垃圾回收的优点

- 1) 悬浮指针（释放后未置空）；
- 2) 重复释放；
- 3) 内存泄露（未释放）。

Java 容器

STL 中数据结构分类

容器

Sequence container: vector deque list

Container adaptor: stack queue priority_queue

Associative container: set multiset map multimap bitset

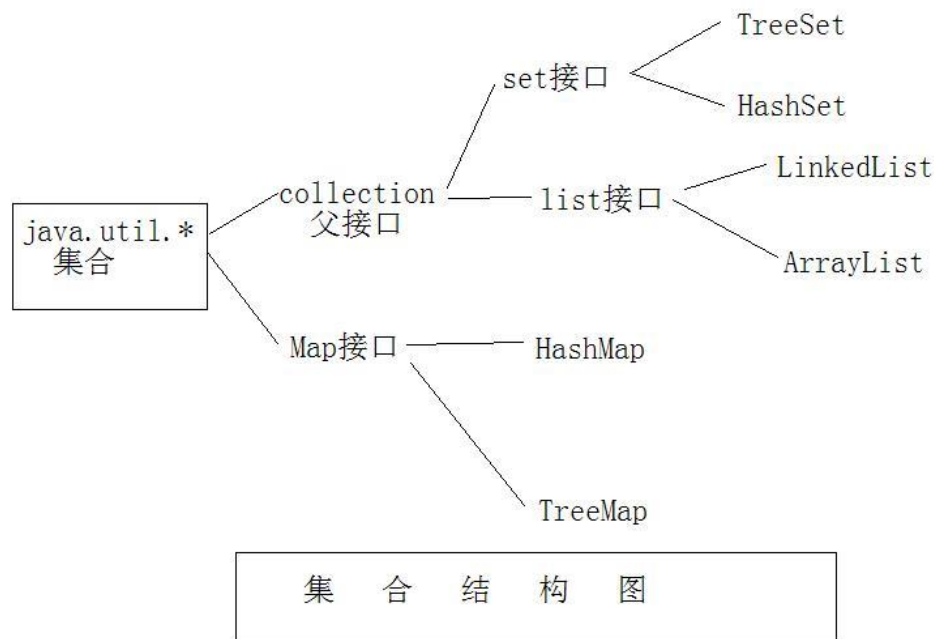
Java 中数据结构分类

java.util.*

Collection 接口 -> List 接口 -> LinkedList / ArrayList

Collection 接口 -> Set 接口 -> TreeSet / HashSet

Map 接口 -> HashMap / TreeMap



Java 反射

在程序运行时动态地创建对象？

举例：1) 通过配置文件创建对象，一个有代表性的例子，Servlet 容器通过 Java 反射 API 来创建所需的 Servlet 实例；2) IDE 通过读取.class 文件自动生成 get/set 方法。

Java 异常

- 1) Error，例如 OutOfMemoryError，程序未执行前，无法确定的错误；
- 2) Unchecked Exception 或者 Runtime Exception，例如 NullPointerException、ArithmeticException、数组越界 IndexOutOfBoundsException、类型转换，**程序员未仔细检查代码而可能产生的异常**；
- 3) Checked Exception，例如 IOException、SQLException、InterruptedException，检查型异常，编译时就能确定，必须要使用 try/catch 语句捕获异常，或者 throw 到上层调用者。

Java 多线程

Thread 类和 Runnable 接口区别

<http://developer.51cto.com/art/201203/321042.htm>

<http://13755101964-163-com.iteye.com/blog/1436277>

在程序开发中只要是多线程肯定永远以实现 Runnable 接口为主，因为实现 Runnable 接口相比继承 Thread 类有如下好处：

- 1) 避免单继承的局限性，一个类可以继承多个接口；

2) 适合于资源的共享（如何理解？让多个线程执行同一个 Runnable 接口实现类的对象）。

Java 对象锁

[java.util.concurrent.locks 与 synchronized 及其异同](#)

Java 拾遗

@Arrays sort(T[] a, c) 函数体 用到 clone 方法（深拷贝）

插入+快速排序，是否保证稳定性？

@数组在 Java 中也是一个类吗？为什么数组会有 Object 类中的方法，还有 length 属性等？

@Java 中的集合类，如何动态分配存储空间？

Java 中的容器类如何自动扩容？依据是什么？

ArrayList 自动扩容为原来的 1.5 倍

HashMap 当容量达到初始化的 0.75 时，自动扩容为原来的 2 倍

ArrayList 中 ensureCapacity()方法增加容量以提高插入效率

@Java 中的排序算法如何实现？

java.util.Arrays 类，

Arrays 类中私有方法 sort1()方法，插入+快排。

sort 和 binarySearch 方法

java.util.Collections 类，

sort(List<T> list) 归并排序？

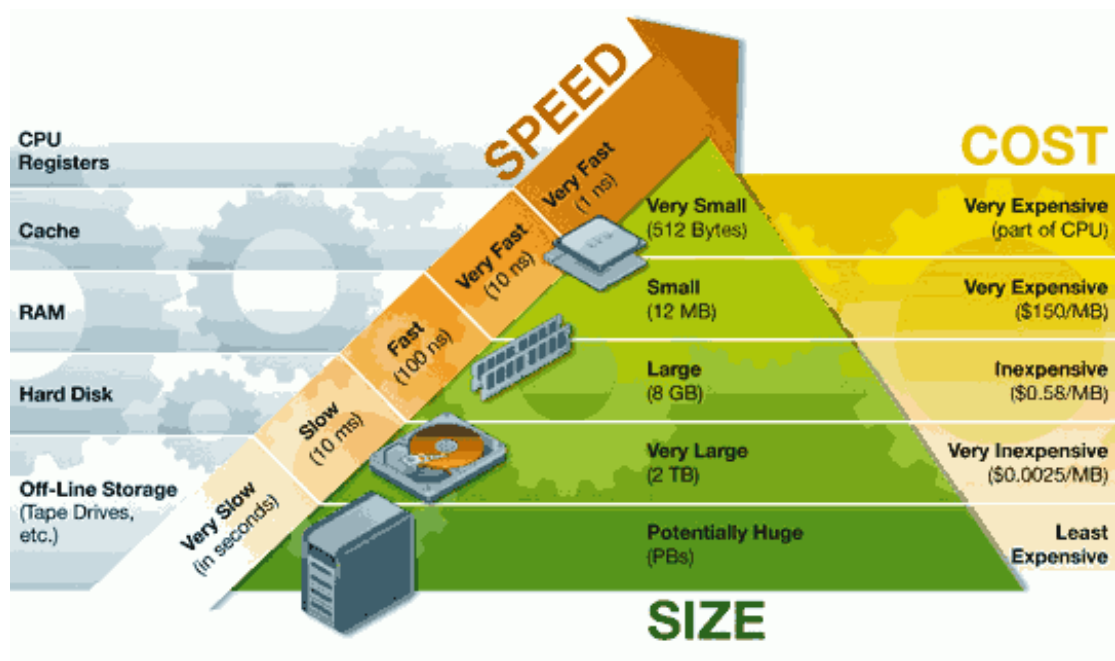
@String 类的所有方法，split 方法

@Java 中构造函数为什么不能是虚函数？抽象类和接口的区别？

@Java 中大数运算如何实现的？数制转换，字符串与整数、浮点数之间的转换如何实现？

10. 大数据/云计算

Hadoop 应用外排序、鸡蛋/篮子问题



华为笔试面试 回顾

共两轮面试

第一轮 技术面

自我介绍，问项目，问为什么选择底层软件开发职位。

两道编程题

第一题

字符串比较问题，要求写出函数实现。虽然简单，注意边界就可以了。

第二题

问题描述：输入一系列成对节点(x_i, y_i)，节点 x_i 是 y_i 的父结点。

接着输入任意一对节点(x, y)，求它们的关系，是否有共同祖先，如果有，隔了几代？

笨方法：构建所有节点存储方式，在孩子节点中设指针指向父结点，祖先节点指向 NULL。

当输入任意一点节点(x, y)，分别沿 x 和 y 上溯，找到各自达到祖先的路径，比较这两条路径。

时间复杂度 $O(n^2)$

面试官给出的方法

构建存储方式：

$[A, C] \rightarrow [C, F] \rightarrow [B, E] \rightarrow [E, H] \rightarrow [F, D]$

输入(A, D)，扫描上面的链表，得到 $A \rightarrow C \rightarrow F \rightarrow D$ ， A 和 D 差 3 代；

输入(A, E)，没有共同祖先

时间复杂度 $O(n)$

提到上机题第三道，因为我答对。

第三道是这样的，以字符串形式输入四则运算表达式，“ $100+50*20/3-1$ ”，求表达式的值。

我说，考试完后有去想到两种方法

- 1) 一个用栈，先计算优先级高的，再计算优先级低的。
- 2) 中缀表达式转后缀表达式，再计算后缀表达式的值。

然后，面试官说，这道题你用有限状态机，秒杀。

面试官点评：你对基本数据结构比较熟悉，但是呢，还缺乏对它们的灵活运用。我这边给你……过。

虚惊一场。

第二轮 综合面试

谈人生，谈理想，谈我对自己了解，有什么特点。谈如何面对压力。谈自己在所选职位上的优势。谈性格，我提到，喜欢独处，也喜欢群居。

面试官中间给我点评：不是一个十分专注的人，有点懒惰，享受生活。

我解释，这是一种自我调节、应对生活压力的方式。

面试官最后给予点评：你的语言表达能力可以，心智成熟，我对你综合能力是认可的。向我表示适合进入华为。

#if 0

```
#include <iostream>
#include <string>
#include <stack>
#include "string.h"
using namespace std;

string infixToSuffix(string infix) {
    stack<char> opStack;
    string suffix;
    for (unsigned i = 0; i < infix.size(); ++i) {
        char cur = infix[i];
        switch (cur) {
            case ' ':
                break;
            case '(':
                opStack.push(cur);
                break;
            case ')':
                while (!opStack.empty() && opStack.top() != '(') {
                    suffix.append(1, ' ');
                    suffix.append(1, opStack.top());
                    opStack.pop();
                }
                opStack.pop();
                break;
            case '+':
            case '-':
                while (!opStack.empty() && opStack.top() != '(') {
                    suffix.append(1, ' ');
                    suffix.append(1, opStack.top());
                    opStack.pop();
                }
                opStack.push(cur);
                break;
            default:
                suffix.append(1, cur);
                break;
        }
    }
    return suffix;
}
```

```

        }
        opStack.push(cur);
        suffix.append(1, ' ');
        break;
    case '*':
    case '/':
        while (!opStack.empty()) {
            char top = opStack.top();
            if (top == '(' || top == '+' || top == '-') {
                break;
            }
            suffix.append(1, ' ');
            suffix.append(1, opStack.top());
            opStack.pop();
        }
        opStack.push(cur);
        suffix.append(1, ' ');
        break;
    default:
        suffix.append(1, cur);
    }
}

while (!opStack.empty()) {
    suffix.append(1, ' ');
    suffix.append(1, opStack.top());
    opStack.pop();
}

return suffix;
}

```

```

bool isOperator(char *s) {
    if (strcmp(s, "+") == 0 ||
        strcmp(s, "-") == 0 ||
        strcmp(s, "*") == 0 ||
        strcmp(s, "/") == 0 ) {
        return true;
    }
    return false;
}

```

```

double calcSuffix(string suffix) {
    stack<double> dStack;
    char *str = _strdup(suffix.c_str());
    char *token;

```

```

char *delim = " ";
for (token = strtok(str, delim); token != NULL; token = strtok(NULL, delim)) {
    if (isOperator(token)) {
        double second = dStack.top();
        dStack.pop();
        double first = dStack.top();
        dStack.pop();
        double temporary;
        if (strcmp(token, "+") == 0) {
            temporary = first + second;
        } else if (strcmp(token, "-") == 0) {
            temporary = first - second;
        } else if (strcmp(token, "*") == 0) {
            temporary = first * second;
        } else if (strcmp(token, "/") == 0) {
            temporary = first / second;
        } else {
            ;
        }
        dStack.push(temporary);
    } else {
        dStack.push(atoi(token));
    }
}
return dStack.top();
}

int main() {
    string infix, suffix;
    while (getline(cin, infix)) {
        suffix = infixToSuffix(infix);
        cout << suffix << endl;
        double result = calcSuffix(suffix);
        cout << "result = " << result << endl;
    }

    return 0;
}

#endif

```

阿里巴巴面试回顾

首先，就算刚赶到笔试现场，没有时间缓和，回答问题应该有条理性，不急不慢。太慌张，会让面试官不舒服的。

1. C++和 Java 的区别

答：它们运行在不同的平台上。C++源代码编译后产生二进制可执行文件，由原生操作系统执行，特例 Qt 具有“write once, compile everywhere”；Java 源代码编译后产生中间字节码，由 Java 虚拟机执行，具有“write once, run anywhere”的特点。C++的程序出错，能够导致系统崩溃；Java 程序出错，顶多是虚拟机崩溃。

C++和 Java 有不同的运行时，也意味着，创建对象的方式是不同的。

2. Java 中的 TreeSet 和 HashSet 的区别？

答：TreeSet 实现 Comparable 接口，存储的数据是非重复的、有序的，有两种方式实现判重：

- 1) 存储元素实现 Comparable 接口，重载 compareTo 方法；
- 2) 定义一个 Comparator，重载 compare 方法，并将该比较器的对象作为参与传给 TreeSet 对象。

HashSet，存储的数据是非重复的、无序的，首先调用 hashCode 方法判重，如果相等，再调用 equals 方法，如果也相等则认为重复。

2.1 什么时候用 TreeSet 或 HashSet 呢？

<http://stackoverflow.com/questions/1463284/hashset-vs-treeset>

答：如果我们需要一个快速的 Set，就选择 HashSet，使用一个哈希表，增、删、查、改的时间复杂度是 $O(1)$ ；如果我们需要一个有序的 Set，就选择 TreeSet，使用一棵红黑树，增、删、查、改的时间复杂度是 $O(\log(n))$ 。

那我想要一个又快又有序的 Set 呢？就选择 LinkedHashSet。

<http://java.dzone.com/articles/hashset-vs-treeset-vs>

Java 数据结构概览

<http://webservices.ctocio.com.cn/java/435/8907435.shtml>

2.2 Java 中线程安全的类

答：StringBuffer、Vector、Hashtable(Properties 的父类，为什么选择从 Hashtable 继承？)，对应线程不安全的分别为 StringBuilder、ArrayList、HashMap。另外，Stack 继承自 Vector，所以 Stack 也是线程安全的。

<http://hnote.org/java-2/stringbuffer-stringbuilder-hashtable-hashmap-arraylist-vector> 区别

2.3 Java 中的队列是对应哪个类？

Queue 是接口，LinkedList 实现 Deque 接口，LinkedList 可以作为队列使用。

<http://blog.csdn.net/nei504293736/article/details/7207380>

<http://blog.csdn.net/javaalpha/article/details/5387017>

3. 有一亿个用户、电话号码、消费，排序，统计每个用户的消费。

哈希，从另一个角度看，就是将一堆数据进行某种程度上的分类

4. Java 阻塞 IO 和非阻塞 IO

5. Java 虚拟机的认识

6. Java 垃圾回收

7. 进程和线程的区别

8. 数组和链表的区别，应该提到操作的时间复杂度，用 $O(1)$ 和 $O(n)$ 来描述

面试之前，路上想的

B 树和 B+树的区别

B 树的数据存储在中间节点和叶子节点上，有利于缓存，即经常访问的数据可以得到更快速

的检索：

B+树的数组都存储在叶子节点，关键字在中间节点上，一次可以将更多的关键字载入内存，此外，叶子节点之间存在链式关系，遍历所有数据非常方便。

很多问题，在网上都能找到原型，说明现在的面试题都是大同小异，应该去网上搜罗一下这些题目。

面试之前，还是应该系统地复习一下已经掌握的知识点。

What is the difference between List, Set and Map?

自己总结，Set 和 List 存储的数据是一元的(unary)，Map 存储的数据是二元的(binary)。

A Set is a collection that has no duplicate elements.

A List is a collection that has an order associated with its elements.

A map is a way of storing key/value pairs. The way of storing a Map is similar to two-column table.

get、post、put 区别

GET safe and idempotent. 安全和幂等

put not safe but idempotent

upload image file ,use put not post... prevent duplicating

post neither safe and idempotent.

POST 不能发送结构化的数据，，所以引入 SOAP。SOAP 基于 XML 协议，可序列化所有对象。

Java 集合框架之小结

<http://jiangzhengjun.iteye.com/blog/553191>

Java 集合容器总结

http://blog.sina.com.cn/s/blog_768c0b4501013qvv.html

回顾 C++ STL 中的数据结构

C++ STL 的实现：

1. vector 底层数据结构为数组，支持快速随机访问
2. list 底层数据结构为双向链表，支持快速增删
3. deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问
4. stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时
5. queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时
6. stack 和 queue 是适配器，而不叫容器，因为是对容器的再封装
7. priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现？
8. set 底层数据结构为红黑树，有序，不重复

9.multiset 底层数据结构为红黑树，有序，可重复
10.map 底层数据结构为红黑树，有序，不重复
11.multimap 底层数据结构为红黑树，有序，可重复
12.hash_set 底层数据结构为 hash 表，无序，不重复
13.hash_multiset 底层数据结构为 hash 表，无序，可重复
14.hash_map 底层数据结构为 hash 表，无序，不重复
15.hash_multimap 底层数据结构为 hash 表，无序，可重复
参考链接: <http://sxnuwhui.blog.163.com/blog/static/1370683732012826102856138/>

腾讯+百度笔试回顾

1. 全排列

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

void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

void permutate(int a[], int n, int j) {
    if (j == n-1) {
        for (int i = 0; i < n; ++i) {
            cout << a[i] << " ";
        }
        cout << endl;
    } else {
        for (int i = j; i < n; ++i) {
            swap(a, i, j);
            permutate(a, n, j+1);
            swap(a, i, j);
        }
    }
}

int main() {
    int a[] = {1, 2, 3, 4};
    permutate(a, 4, 0);
    return 0;
}
```

2. 给定一个数组 `int a[] = {4, 5, 4, 3, 2, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 9}`; 数组相邻元素的差的绝对值为 1, 输入数组 a 和待查找整数 t, 求 t 在数组 a 中位置。

```
int search(int a[], int n, int t) {
    int i = 0;
    while (i < n && a[i] != t) {
        i += abs(t - a[i]);
    }
    if (i < n) {
        return i;
    }
    return -1;
}
```

3. 一棵二叉树的高度为最深的叶子节点到跟节点的路径长度, 此二叉树的宽度为某层节点数最大值, 求此二叉树的面积。

```
#include <iostream>
#include "math.h"
#include <queue>
#include "string.h"
using namespace std;
```

```
struct TreeNode {
    int data;
    TreeNode *left;
    TreeNode *right;
    int level;
};
```

```
void createTree(TreeNode **root) {
    *root = new TreeNode();
    (*root)->left = new TreeNode();
    (*root)->right = new TreeNode();
    (*root)->left->left = new TreeNode();
    (*root)->left->right = new TreeNode();
    (*root)->left->left->left = new TreeNode();
    (*root)->left->left->right = new TreeNode();
    (*root)->left->left->left->left = new TreeNode();
    (*root)->left->left->left->left->left = new TreeNode();
    (*root)->right->right = new TreeNode();
}
```

```
void clearTree(TreeNode *root) {
    if (root != NULL) {
        clearTree(root->left);
```

```

        clearTree(root->right);
        cout << "clear " << &root << endl;
        delete root;
    }
}

int height(const TreeNode *root) {
    if (root != NULL) {
        int left = height(root->left);
        int right = height(root->right);
        return (left > right ? left : right) + 1;
    }
    return 0;
}

int Area(TreeNode *root) {
    int depth = height(root); //计算二叉树的层数
    cout << "深度" << depth << endl;
    int *count = new int[depth]; //每一层都一个计数器
    memset(count, 0, depth*sizeof(count));
    root->level = 0;
    queue<TreeNode*> Q;
    Q.push(root);
    while (!Q.empty()) {
        TreeNode *cur = Q.front(); Q.pop();
        count[cur->level]++;
        if (cur->left != NULL) {
            cur->left->level = cur->level + 1;
            Q.push(cur->left);
        }
        if (cur->right != NULL) {
            cur->right->level = cur->level + 1;
            Q.push(cur->right);
        }
    }
    int width = 0;
    for (int i = 0; i < depth; ++i) {
        if (count[i] > width) {
            width = count[i];
        }
    }
    cout << "宽度" << width << endl;
    return depth * width;
}

```



```

int main() {
    TreeNode *root;
    createTree(&root);
    Area(root);
    clearTree(root);
    return 0;
}

```

4. TCP 和 UDP 的区别

TCP 是一种面向连接的、可靠的、基于字节流的传输层协议，具有**差错检测**、流量控制和拥塞控制功能（error detection、flow control、congestion control）。基于 TCP 的上层协议有 HTTP、TLS、POP3（客户端与邮件服务器）、SMTP（邮件服务器之间）、FTP 等。

UDP 是一种面向非连接的、不可靠的、基于数据报的传输层协议。基于 UDP 的上层协议有 DNS（域名解析）、RTP（多媒体，看视频）、SNMP。

HTTP 的特点？

TLS 安全机制？

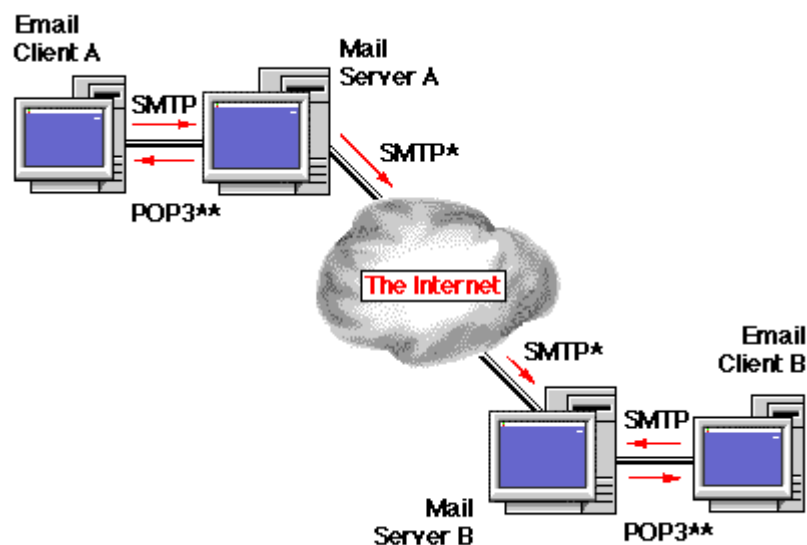
FTP 的工作原理？

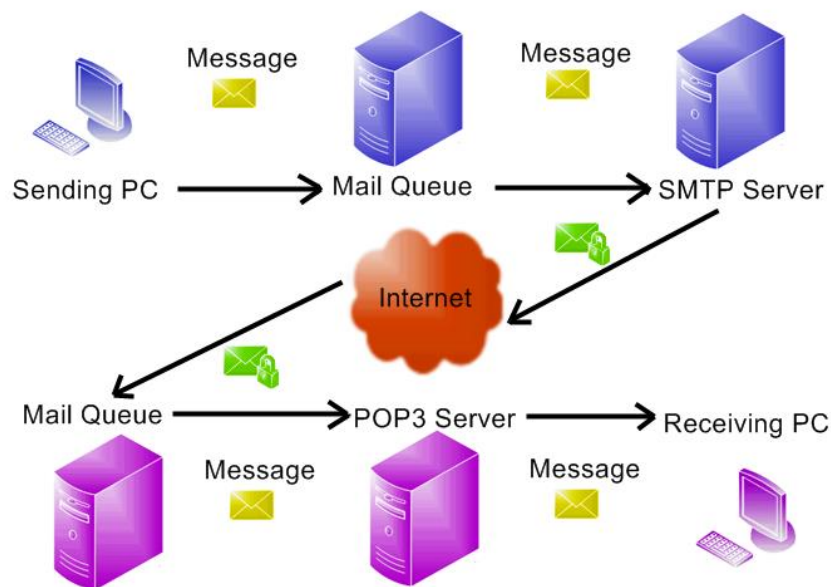
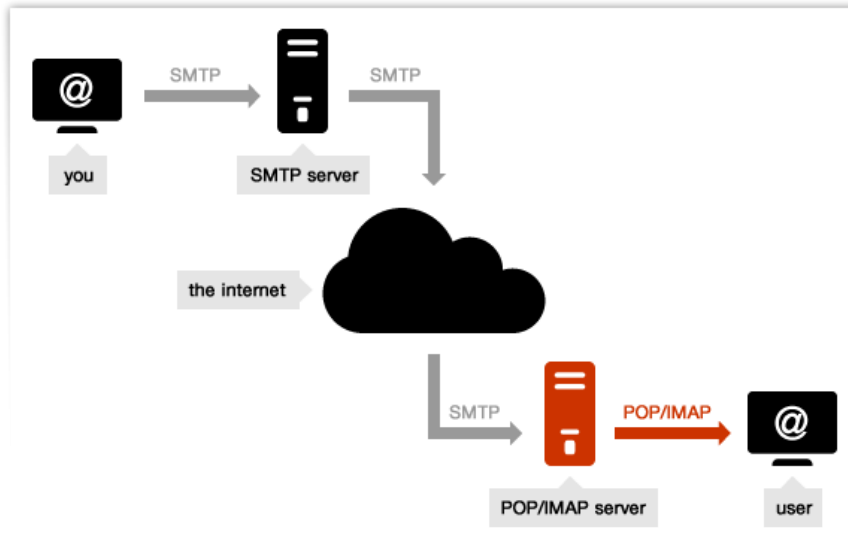
DNS 的工作原理？

SNMP 的用途？

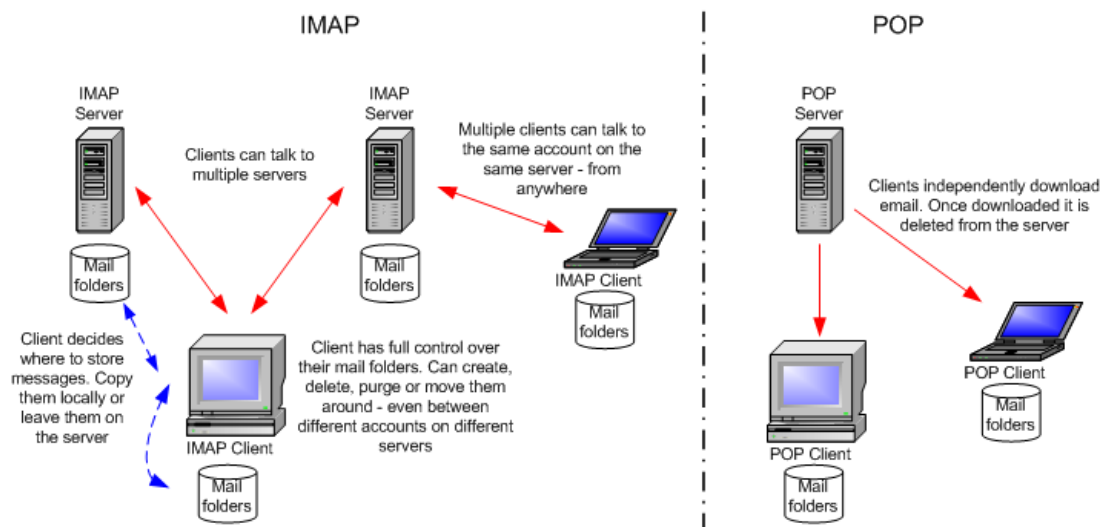
4.1 POP3 与 SMTP 的区别？

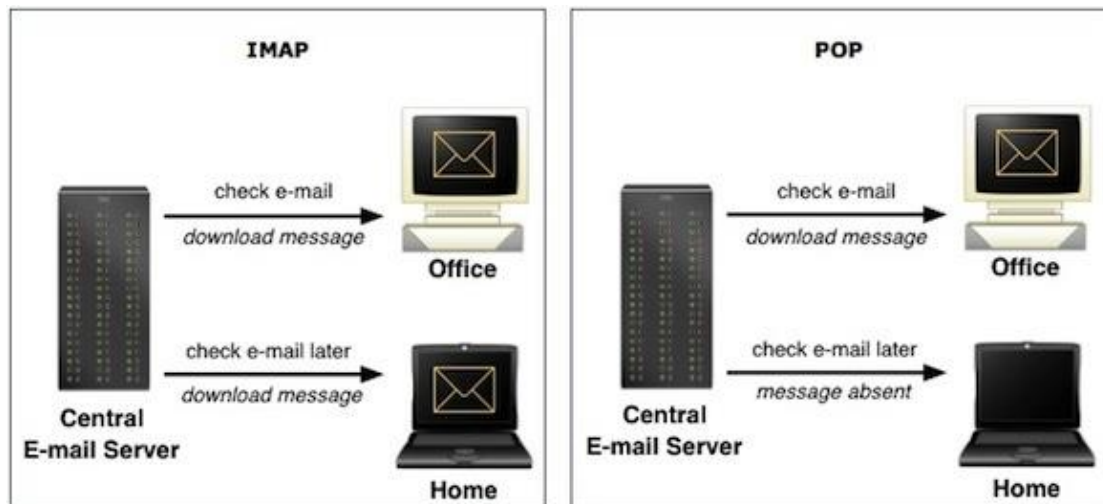
POP is a protocol for storage of email. SMTP is a protocol for sending and receiving email.





POP3 与 SMTP 数据流程图

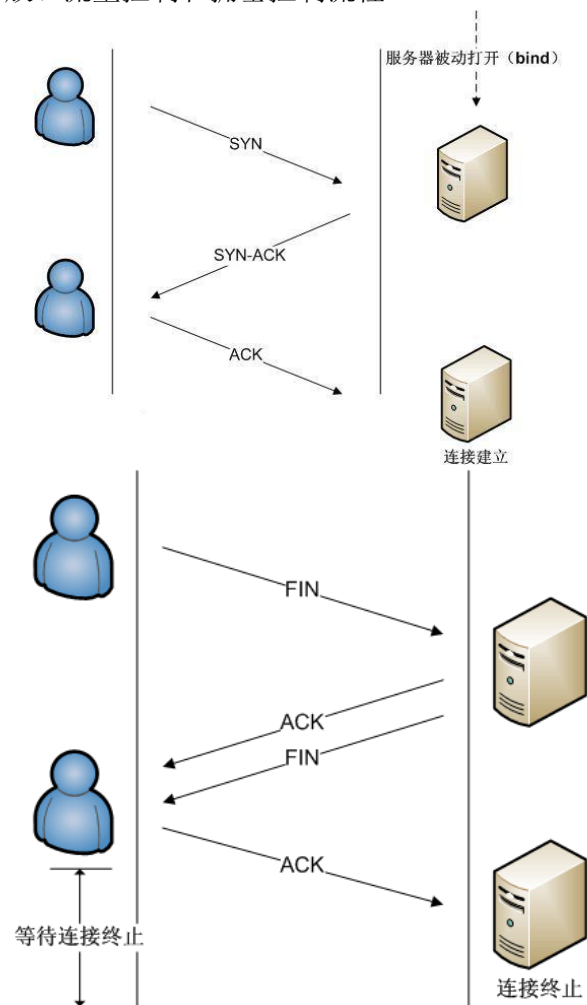




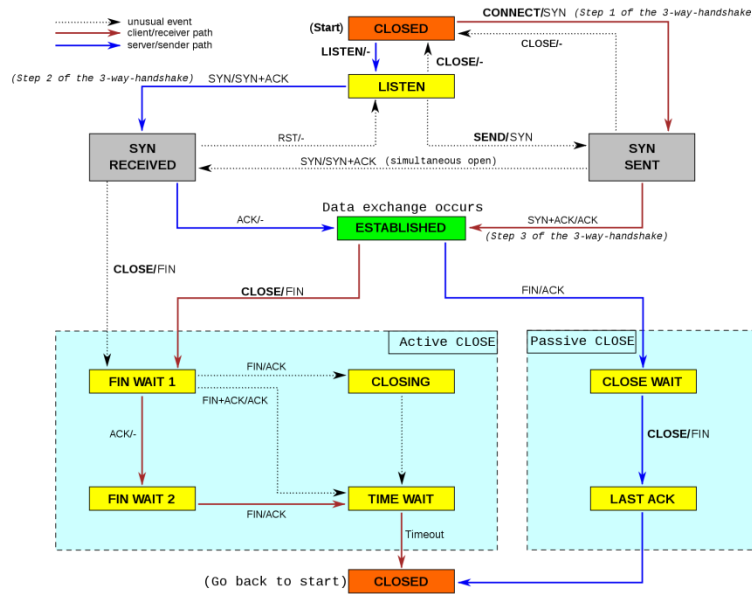
the difference of POP3 and IMTP(Internet Mail Access Protocol)

从今天开始，习惯使用图片搜索，深刻体会图片表达的强大！

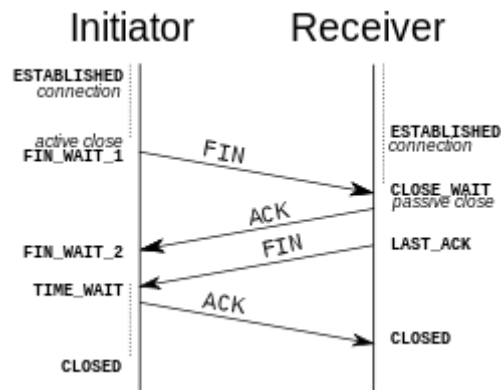
4.2 TCP 的建立、释放、流量控制和拥塞控制流程？



TCP 连接的建立和释放



TCP 状态机



5. 可重入函数

在 `read`、`time`、`localtime` 和 `new` 函数中，只有 `new` 函数是可重入的。

若一个函数是可重入的，则该函数：

- 不能含有静态（全局）非常量数据。
- 不能返回静态（全局）非常量数据的地址。
- 只能处理由调用者提供的数据。
- 不能依赖于单实例模式资源的锁。
- 不能调用(call)不可重入的函数(有调用到的函数需满足前述条件)。

多“用户/对象/进程优先级”以及多进程，一般会使得对可重入代码的控制变得复杂。同时，IO 代码通常不是可重入的，因为他们依赖于像磁盘这样共享的、单独的(类似编程中的静态(Static)、全域(Global)资源)。

6. 判断两个矩形是否相交

假定矩形是用一对点表达的(minx, miny) (maxx, maxy)，那么两个矩形

`rect1{(minx1, miny1)(maxx1, maxy1)}`

```
rect2{(minx2, miny2)(maxx2, maxy2)}
```

相交的结果一定是一个矩形, 构成这个相交矩形 `rect{(minx, miny) (maxx, maxy)}` 的点对坐标是:

```
minx   =   max(minx1,   minx2)
miny   =   max(miny1,   miny2)
maxx   =   min(maxx1,   maxx2)
maxy   =   min(maxy1,   maxy2)
```

如果两个矩形不相交, 那么计算得到的点对坐标必然满足:

(minx > maxx) 或者 (miny > maxy)

判定是否相交, 以及相交矩形是什么都可以用这个方法一体计算完成。

从这个算法的结果上, 我们还可以简单的生成出下面的两个内容:

(一) 相交矩形: (minx, miny) (maxx, maxy)

(二) 面积: 面积的计算可以和判定一起进行

```
if ( minx>maxx ) return 0;
if ( miny>maxy ) return 0;
return (maxx-minx)*(maxy-miny)
```

参考链接: <http://www.cnblogs.com/0001/archive/2010/05/04/1726905.html>

7. 求一棵二叉树任意两个节点的最近公共祖先

```
#include <iostream>
#include "math.h"
#include <queue>
#include "string.h"
using namespace std;
```

```
struct TreeNode {
    int data;
    TreeNode *left;
    TreeNode *right;
    int level;
    TreeNode(int data = 0, TreeNode *left = NULL, TreeNode *right = NULL, int
level = 0) {
        this->data = data;
        this->left = left;
        this->right = right;
        this->level = level;
    }
};
```

```
void createTree(TreeNode **root) {
```

```

    *root = new TreeNode(1);
    (*root)->left = new TreeNode(2);
    (*root)->right = new TreeNode(3);
    (*root)->left->left = new TreeNode(4);
    (*root)->left->right = new TreeNode(5);
    (*root)->left->left->left = new TreeNode(6);
    (*root)->left->left->right = new TreeNode(7);
    (*root)->left->left->left->left = new TreeNode(8);
    (*root)->left->left->left->left->left = new TreeNode(9);
    (*root)->right->right = new TreeNode(10);
}

void clearTree(TreeNode *root) {
    if (root != NULL) {
        clearTree(root->left);
        clearTree(root->right);
        cout << "clear " << &root << endl;
        delete root;
    }
}

int findLCA(TreeNode *root, TreeNode *a, TreeNode *b, TreeNode **lca) {
    //根节点为空，自不必说，终止递归查找
    //当*lca不为空，说明已经找到最近公共祖先，可进行剪枝
    if (root == NULL || *lca != NULL) {
        return 0;
    }
    int flag = 0, nLen = 0;
    if (root == a || root == b) {
        flag = 1;
    }
    nLen += flag;
    nLen += findLCA(root->left, a, b, lca);
    nLen += findLCA(root->right, a, b, lca);
    if (nLen == 2 && flag == 0 && *lca == NULL) {
        *lca = root;
    }
    return nLen;
}

int main() {
    TreeNode *root;
    createTree(&root);
}

```

```

TreeNode *a = root->left->left;
TreeNode *b = root->left->left->left->left;

printf(" a->data = %d\n", a->data);
printf(" b->data = %d\n", b->data);

TreeNode *out = NULL;
int ret = findLCA(root, a, b, &out);
if (ret == 2) {
    printf("the least common ancestor = %p, %d\n", out, out->data);
} else {
    printf("no LCA!\n");
}
clearTree(root);
return 0;
}

```

算法时间复杂度 $f(n) = \sum_{k=1}^h (2^k \cdot 2^{h-k} + 2^{k-1}) = O(n \log_2 n)$

参考网址:

http://blog.csdn.net/piaojun_pj/article/details/5971125?reload#reply

http://blog.csdn.net/v_july_v/article/details/11921021

8. int &*p 和 int*&p 的区别

int &*p 指向引用的指针是非法的;

int *&p, 实际上可以这么看 int* &p, 即&p 表示一个引用变量, 其引用变量的类型为*int。

6 大排序算法

```

#include <iostream>
#include "stdlib.h"
#include "time.h"
using namespace std;

```

```

void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

```

//冒泡排序

```

void bubbleSort(int a[], int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = n - 1; j > i; --j) {

```

```

        if (a[j] < a[j-1]) {
            swap(a, j, j - 1);
        }
    }
}

//插入排序
void insertSort(int a[], int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j > 0; --j) {
            if (a[j] < a[j-1]) {
                swap(a, j, j - 1);
            }
        }
    }
}

//选择排序
void selectSort(int a[], int n) {
    for (int i = 0; i < n; ++i) {
        int min = i;
        for (int j = i + 1; j < n; ++j) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        if (i != min) {
            swap(a, i, min);
        }
    }
}

//快速排序
void quickSort(int a[], int left, int right) {
    int pivotValue = a[left];
    int pivotIndex = left;
    int i = left, j = right;
    while (i <= j) {
        while (j >= pivotIndex && a[j] >= pivotValue) --j;
        if (j >= pivotIndex) {
            a[pivotIndex] = a[j];
            pivotIndex = j;
        }
        while (i <= pivotIndex && a[i] <= pivotValue) ++i;
        if (i <= pivotIndex) {
            a[pivotIndex] = a[i];

```



```

        pivotIndex = i;
    }
}
a[pivotIndex] = pivotValue;
if (left < pivotIndex - 1) {
    quickSort(a, left, pivotIndex - 1);
}
if (pivotIndex + 1 < right) {
    quickSort(a, pivotIndex + 1, right);
}
}

```

```

void adjustDown(int a[], int root, int leaf) {
    int parent = root;
    int child = 2 * parent + 1;
    while (child <= leaf) {
        if (child < leaf && a[child] < a[child + 1]) {
            child++;
        }
        if (a[parent] >= a[child]) {
            break;
        } else {
            int temp = a[parent];
            a[parent] = a[child];
            a[child] = temp;
            parent = child;
            child = 2 * parent + 1;
        }
    }
}
}

```

```

void createHeap(int a[], int root, int leaf) {
    for (int i = (leaf - 1) / 2; i >= root; --i) {
        adjustDown(a, i, leaf);
    }
}

```

//堆排序

```

void heapSort(int a[], int root, int leaf) {
    createHeap(a, root, leaf);
    for (int i = root; i <= leaf; ++i) {
        int temp = a[root];
        a[root] = a[leaf - i];
        a[leaf - i] = temp;
        adjustDown(a, root, leaf - i - 1);
    }
}

```

```

    }
}

void merge(int a[], int left, int mid, int right) {
    int *b = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right) {
        if (a[i] <= a[j]) {
            b[k++] = a[i++];
        } else {
            b[k++] = a[j++];
        }
    }
    while (i <= mid) {
        b[k++] = a[i++];
    }
    while (j <= right) {
        b[k++] = a[j++];
    }
    for (int i = 0; i < k; ++i) {
        a[left + i] = b[i];
    }
    delete[] b;
}

```

//归并排序

```

void mergeSort(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(a, left, mid);
        mergeSort(a, mid + 1, right);
        merge(a, left, mid, right);
    }
}

```

```

void print(int a[], int n) {
    for (int i = 0; i < n; ++i) {
        cout << a[i] << " ";
    }
    cout << endl;
}

```

```

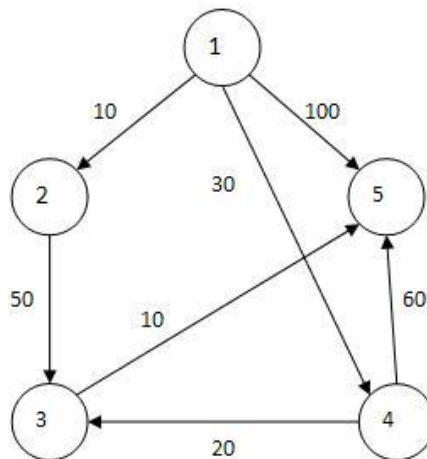
int main() {
    int SIZE = 30;
    int arr[SIZE];
}

```

```

srand((unsigned)time(NULL));
for (int i = 0; i < SIZE; ++i) {
    arr[i] = rand() % SIZE;
}
print(arr, SIZE);
//bubbleSort(arr, SIZE);
//insertSort(arr, SIZE);
//selectSort(arr, SIZE);
//quickSort(arr, 0, SIZE - 1);
//heapSort(arr, 0, SIZE - 1);
mergeSort(arr, 0, SIZE - 1);
print(arr, SIZE);
return 0;
}

```



迭代	S	u	Distance[2]	Distance[3]	Distance[4]	Distance[5]
初始	{1}	-	10	无穷大	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

单源最短路径(Dijkstra 和 Bellman-Ford)

Dijkstra(graph, source)

```

for i = 0 to n-1 do
    dist[i] = weight(source, i)
    if dist[i] == infinity
        prev[i] = -1;
    else
        prev[i] = source;
    visited[i] = false;
visited[source] = true;
for k = 1 to n-1 do

```

```

u = index of the unvisited and smallest one in dist[]
for each unvisited vertex v:
    if dist[v] > dist[u] + weight(u, v)
        dist[v] = dist[u] + weight(u, v);
        prev[v] = u;

```

Bellman-Ford

Bellman-Ford(Graph, source)

```

for i = 0 to n-1 do
    dist[i] = infinity
    prev[i] = -1
dist[source] = 0
for k = 1 to n-1 do
    for each (u, v) in edges of Graph do
        if dist[u] + weight(u, v) < dist[v] do
            dist[v] = dist[u] + weight(u, v)
            prev[v] = u
    for each (u, v) in edges of Graph do
        if dist[u] + weight(u, v) < dist[v] do
            print "no shortest path"

```

最小生成树

口述：单链表反转

链表中，除了首尾两个节点的其它节点都有一个前驱和后继。
遍历链表，设法让当前节点的指针域指向它的前驱。

prev = null, cur = head, next = cur->link;

```

while(cur) {
    cur->link = prev;
    prev = cur;
    cur = next;
    next = cur->link;
}

```

口述：最大子序列和

设 $b[j]$ 表示第 j 处，以 $a[j]$ 结尾的子序列的最大和。

注意： $b[j]$ 并不是前 $j-1$ 个数中最大的连续子序列之和，而只是包含 $a[j]$ 的最大连续子序列的和。我们求出 $b[j]$ 中的最大值，即为所求的最大连续子序列的和。

递推公式

$b[0] = a[0]$ $i = 0$

$b[i] = \max(a[i], b[i-1] + a[i]) \quad i > 0$
 $\max\text{-sum} = \max \{ b[i] \mid 0 \leq i < n \}$

图算法总结

图的定义：

很简单， $G(V, E)$ ， V 、 E 分别表示点和边的集合。

图的表示：

主要有两种，邻接矩阵和邻接表，前者空间复杂度， $O(V^2)$ ，后者为 $O(V+E)$ 。因此，除非非常稠密的图(边非常多)，一般后者优越于前者。

图的遍历：

宽度遍历 BFS(start): (1) 队列 $Q = \text{Empty}$ ，数组 $\text{bool visited}[V] = \{\text{false} \dots\}$. $Q.\text{push}(\text{start})$;

(2) while ($!Q.\text{empty}()$){

$u = Q.\text{pop}()$; $\text{visited}[u] = \text{true}$; //遍历 u 结点

foreach (u 的每一个邻接结点 v) $Q.\text{push}(v)$;

}

深度遍历 DFS(start): (1) 栈 $S = \text{Empty}$ ，数组 $\text{bool visited}[V] = \{\text{false} \dots\}$. $S.\text{push}(\text{start})$;

(2) while ($!S.\text{empty}()$){

$u = S.\text{pop}()$;

if ($!\text{visited}[u]$) $\text{visited}[u] = \text{true}$; //遍历 u 结点

foreach (u 的每一个邻接结点 v) $S.\text{push}(v)$;

}

初看之下两个算法很相似，主要区别在于一个使用队列，一个使用栈，最终导致了遍历的顺序截然不同。队列是先入先出，所以访问 u 以后接下来就访问 u 中未访问过的邻接结点；而栈的后进先出，当访问 u 后，压入了 u 的邻接结点，在后面的循环中，首先访问 u 的第一个邻接点 v ，接下来又将 v 的邻接点 w 压入 S ，这样接下来要访问的自然就是 w 了。

最小生成树：

(1) Prime 算法: (1) 集合 $\text{MST} = T = \text{Empty}$ ，选取 G 中一结点 u ， $T.\text{add}(u)$

(2) 循环 $|V|-1$ 次：

选取一条这样的边 $e = \min\{(x, y) \mid x \in T, y \in V/T\}$

$T.\text{add}(y)$; $\text{MST}.\text{add}(e)$;

(3) MST 即为所求

(2) Kruskal 算法 (1) 将 G 中所有的边排序并放入集合 H 中，初始化集合 $\text{MST} = \text{Empty}$ ，初始化不相交集 $T = \{\{v_1\}, \{v_2\}, \dots\}$ ，也即 T 中每个点为一个集合。

(2) 依次取 H 中的最短边 $e(u, v)$ ，如果 $\text{Find-Set}(u) \neq \text{Find-Set}(v)$ (也即 u 、 v 是否已经在一棵树中)，那么 $\text{Union}(u, v)$ (即 u, v 合并为一个集合)， $\text{MST}.\text{add}(e)$;

(3) MST 即为所求

这两个算法都是贪心算法，区别在于每次选取边的策略。证明该算法的关键在于一点：如果 MST 是图 G 的最小生成树，那么在子图 G' 中包含的子生成树 MST' 也必然是 G' 的最小

生成树。这个很容易反正，假设不成立，那么 G' 有一棵权重和更小的生成树，用它替换掉 MST' ，那么对于 G 我们就找到了比 MST 更小的生成树，显然这与我们的假设(MST 是最小生成树)矛盾了。

理解了这个关键点，算法的正确性就好理解多了。对于 Prime， T 于 V/T 两个点集都会各自有一棵生成树，最后要连起来构成一棵大的生成树，那么显然要选两者之间的最短的那条边了。对于 Kruskal 算法，如果当前选取的边没有引起环路，那么正确性是显然的（对给定点集依次选最小的边构成一棵树当然是最小生成树了），如果导致了环路，那么说明两个点都在该点集里，由于已经构成了树（否则也不可能导致环路）并且一直都是挑尽可能小的，所以肯定是最小生成树。

最短路径:

这里的算法基本是基于动态规划和贪心算法的，经典算法有很多个，主要区别在于：有的是通用的，有的是针对某一类图的，例如，无负环的图，或者无负权边的图等。

单源最短路径（1）通用（Bellman-Ford 算法）:

（2）无负权边的图(Dijkstra 算法):

（3）无环有向图(DAG) :

所有结点间最短路径:

（1）Floyd-Warshall 算法:

（2）Johnson 算法:

关键路径:

又一个很讨巧的做法，把所有边的权重取反，然后求最短路径。

拓扑排序:

二叉树的非递归遍历

<http://www.cnblogs.com/dolphin0520/archive/2011/08/25/2153720.html>

C++11 标准 右值引用

<http://www.cnblogs.com/soaliap/archive/2012/11/19/2777131.html>

<http://blog.csdn.net/srzhz/article/details/7921546>

<http://zh.wikipedia.org/wiki/C%2B%2B11>

http://coolshell.cn/articles/10478.html?utm_campaign=Manong_Weekly_Issue_6&utm_medium=EDM&utm_source=Manong_Weekly

二叉树非递归中序遍历

提示：二叉树本身是递归定义的，紧抓根节点、左子树和右子树的关系。

思考:

root

left right

a b c d

中序遍历就是先遍历 root 的左子树，再遍历 root，最后遍历 root 的右子树，并且这个过程是递归地进行。

探测顺序：root->left->a->NULL（a 最后探测，要最先访问，所以这些节点要入栈）a 的左子

树为空，这时就要让 **a** 的右子树也入栈，结果 **a** 的右子树也为空，则从栈中 **pop** 出一个元素 **t**，并访问它，由于 **t** 的左子树之前已近被访问过，所以要访问 **t** 的右子树。

```
void nrInOrder(TreeNode *root) {
    TreeNode *p = root;
    stack<TreeNode*> S;
    while (p || !S.empty()) {
        while (p) {
            S.push(p);
            p = p->left;
        }
        if (!S.empty()) {
            TreeNode *t = S.top(); S.pop();
            cout << t->data << " ";
            p = t->right;
        }
    }
}
```

同样先序遍历的探测顺序

root[*]->left[*]->a->NULL->left->b...

```
void nrPreOrder(TreeNode *root) {
    TreeNode *p = root;
    stack<TreeNode*> S;
    while (p || !S.empty()) {
        while (p) {
            cout << p->data << " ";
            S.push(p);
            p = p->left;
        }
        if (!S.empty()) {
            TreeNode *t = S.top(); S.pop();
            p = t->right;
        }
    }
}
```

后续遍历

有没有比较好的方法？

后序遍历的非递归实现是三种遍历方式中最难的一种。因为在后序遍历中，要保证左孩子和右孩子都已被访问并且左孩子在右孩子前访问才能访问根结点，这就为流程的控制带来了难题。下面介绍两种思路。

第一种思路：对于任一结点 P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问，因此其右孩子还未被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多设置一个变量标识该结点是否是第一次出现在栈顶。




```

void postOrder2(BinTree *root)    //非递归后序遍历
{
    stack<BTNode*> s;
    BinTree *p=root;
    BTNode *temp;
    while(p!=NULL || !s.empty())
    {
        while(p!=NULL)            //沿左子树一直往下搜索，直至出现没有左子树的结点
        {
            BTNode *btn=(BTNode *)malloc(sizeof(BTNode));
            btn->btnode=p;
            btn->isFirst=true;
            s.push(btn);
            p=p->lchild;
        }
        if(!s.empty())
        {
            temp=s.top();
            s.pop();
            if(temp->isFirst==true)    //表示是第一次出现在栈顶
            {
                temp->isFirst=false;
                s.push(temp);
                p=temp->btnode->rchild;
            }
            else                    //第二次出现在栈顶
            {
                cout<<temp->btnode->data<<" ";
                p=NULL;
            }
        }
    }
}


```



第二种思路：要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点 P，先将其入栈。如果 P 不存在左孩子和右孩子，则可以直接访问它；或者 P 存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将 P 的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。



```
void postOrder3(BinTree *root)    //非递归后序遍历
{
    stack<BinTree*> s;
    BinTree *cur;                //当前结点
    BinTree *pre=NULL;           //前一次访问的结点
    s.push(root);
    while(!s.empty())
    {
        cur=s.top();
        //红色部分表示 cur 的左孩子或右孩子已经被访问过
        if((cur->lchild==NULL&&cur->rchild==NULL) ||
            (pre!=NULL&&(pre==cur->lchild || pre==cur->rchild)))
        {
            cout<<cur->data<<" "; //如果当前结点没有孩子结点或者孩子
            节点都被访问过
            s.pop();
            pre=cur;
        }
        else
        {
            if(cur->rchild!=NULL)
                s.push(cur->rchild);
            if(cur->lchild!=NULL)
                s.push(cur->lchild);
        }
    }
}
```



阿里巴巴

首先，就算刚赶到笔试现场，没有时间缓和，回答问题应该有条理性，不急不慢。太慌张，会让面试官不舒服的。

1. C++和 Java 的区别

答：它们运行在不同的平台上。C++源代码编译后产生二进制可执行文件，由原生操作系统执行，特例 Qt 具有“write once, compile everywhere”；Java 源代码编译后产生中间字节码，由 Java 虚拟机执行，具有“write once, run anywhere”的特点。C++的程序出错，能够导致系统崩溃；Java 程序出错，顶多是虚拟机崩溃。

C++和 Java 有不同的运行时，也意味着，创建对象的方式是不同的。

2. Java 中的 TreeSet 和 HashSet 的区别？

答：TreeSet 实现 Comparable 接口，存储的数据是非重复的、有序的，有两种方式实现判重：

- 1) 存储元素实现 Comparable 接口，重载 compareTo 方法；
- 2) 定义一个 Comparator，重载 compare 方法，并将该比较器的对象作为参与传给 TreeSet 对象。

HashSet，存储的数据是非重复的、无序的，首先调用 hashCode 方法判重，如果相等，再调用 equals 方法，如果也相等则认为重复。

2.1 什么时候用 TreeSet 或 HashSet 呢？

<http://stackoverflow.com/questions/1463284/hashset-vs-treeset>

答：如果我们需要一个快速的 Set，就选择 HashSet，使用一个哈希表，增、删、查、改的时间复杂度是 $O(1)$ ；如果我们需要一个有序的 Set，就选择 TreeSet，使用一棵红黑树，增、删、查、改的时间复杂度是 $O(\log(n))$ 。

那我想要一个又快又有序的 Set 呢？就选择 LinkedHashSet。

<http://java.dzone.com/articles/hashset-vs-treeset-vs>

Java 数据结构概览

<http://webservices.ctocio.com.cn/java/435/8907435.shtml>

2.2 Java 中线程安全的类

答：StringBuffer、Vector、Hashtable(Properties 的父类，为什么选择从 Hashtable 继承？)，对应线程不安全的分别为 StringBuilder、ArrayList、HashMap。另外，Stack 继承自 Vector，所以 Stack 也是线程安全的。

<http://hnote.org/java-2/stringbuffer-stringbuilder-hashtable-hashmap-arraylist-vector> 区别

2.3 Java 中的队列是对应哪个类？

Queue 是接口，LinkedList 实现 Deque 接口，LinkedList 可以作为队列使用。

<http://blog.csdn.net/nei504293736/article/details/7207380>

<http://blog.csdn.net/javaalpha/article/details/5387017>

3. 有一亿个用户、电话号码、消费，排序，统计每个用户的消费。

哈希，从另一个角度看，就是将一堆数据进行某种程度上的分类

4. Java 阻塞 IO 和非阻塞 IO

5. Java 虚拟机的认识

6. Java 垃圾回收

7. 进程和线程的区别

8. 数组和链表的区别，应该提到操作的时间复杂度，用 $O(1)$ 和 $O(n)$ 来描述

面试之前，路上想的

B 树和 B+树的区别

B 树的数据存储在中间节点和叶子节点上，有利于缓存，即经常访问的数据可以得到更快速的检索；

B+树的数组都存储在叶子节点，关键字在中间节点上，一次可以将更多的关键字载入内存，此外，叶子节点之间存在链式关系，遍历所有数据非常方便。

很多问题，在网上都能找到原型，说明现在的面试题都是大同小异，应该去网上搜罗一下这些题目。

面试之前，还是应该系统地复习一下已经掌握的知识。

What is the difference between List, Set and Map?

自己总结，Set 和 List 存储的数据是一元的(unary)，Map 存储的数据是二元的(binary)。

A Set is a collection that has no duplicate elements.

A List is a collection that has an order associated with its elements.

A map is a way of storing key/value pairs. The way of storing a Map is similar to two-column table.

get、post、put 区别

GET safe and idempotent. 安全和幂等

put not safe but idempotent

upload image file ,use put not post... prevent duplicating

post neither safe and idempotent.

POST 不能发送结构化的数据，，所以引入 SOAP。SOAP 基于 XML 协议，可序列化所有对象。

Java 集合框架之小结

<http://jiangzhengjun.iteye.com/blog/553191>

Java 集合容器总结

http://blog.sina.com.cn/s/blog_768c0b4501013qww.html

回顾 C++ STL 中的数据结构

C++ STL 的实现：

1. vector 底层数据结构为数组，支持快速随机访问
2. list 底层数据结构为双向链表，支持快速增删
3. deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问
4. stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时
5. queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时
6. stack 和 queue 是适配器，而不叫容器，因为是对容器的再封装

7. `priority_queue` 的底层数据结构一般为 `vector` 为底层容器，堆 `heap` 为处理规则来管理底层容器实现？

8. `set` 底层数据结构为红黑树，有序，不重复

9. `multiset` 底层数据结构为红黑树，有序，可重复

10. `map` 底层数据结构为红黑树，有序，不重复

11. `multimap` 底层数据结构为红黑树，有序，可重复

12. `hash_set` 底层数据结构为 hash 表，无序，不重复

13. `hash_multiset` 底层数据结构为 hash 表，无序，可重复

14. `hash_map` 底层数据结构为 hash 表，无序，不重复

15. `hash_multimap` 底层数据结构为 hash 表，无序，可重复

参考链接：<http://sxnuwhui.blog.163.com/blog/static/1370683732012826102856138/>

java 集合(容器)总结

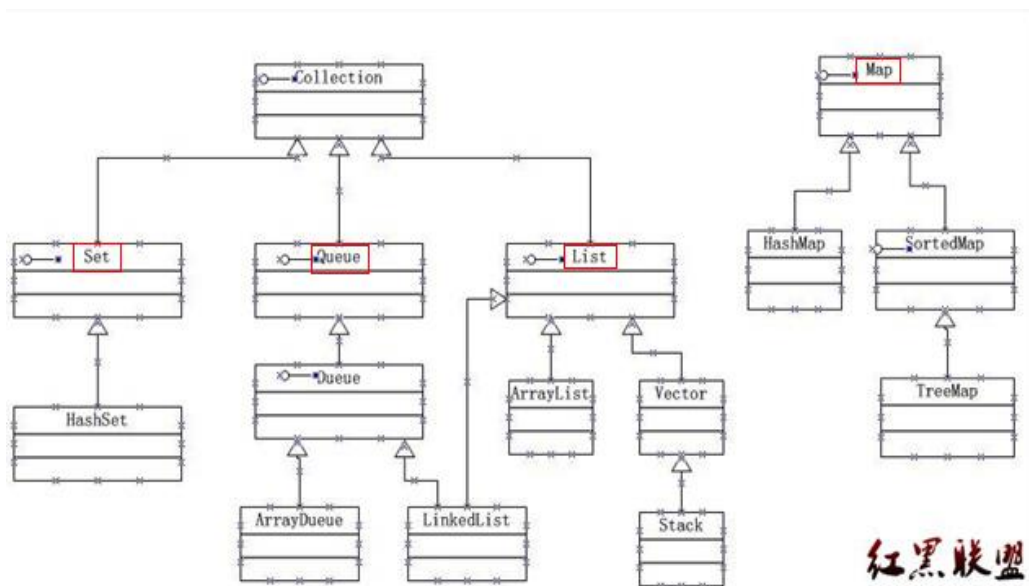
1. 种类: set list map queue 接口

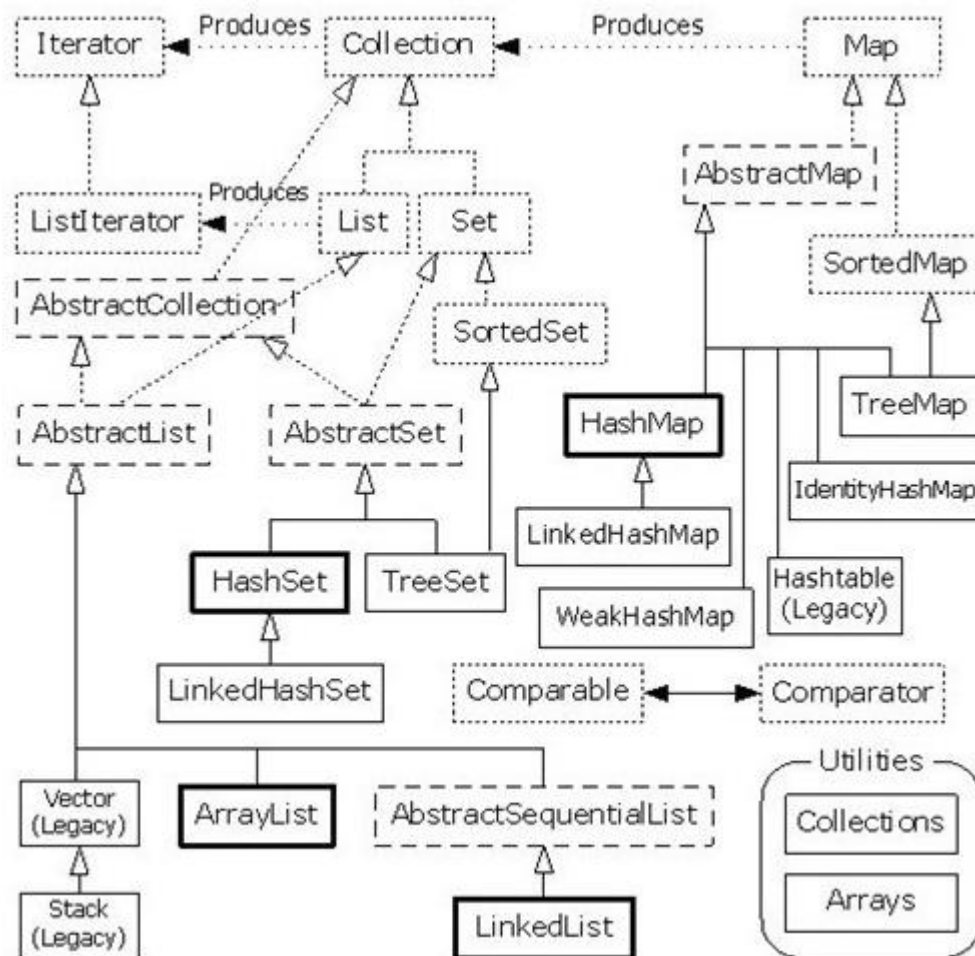
<1> `set`: 无序且不可重复

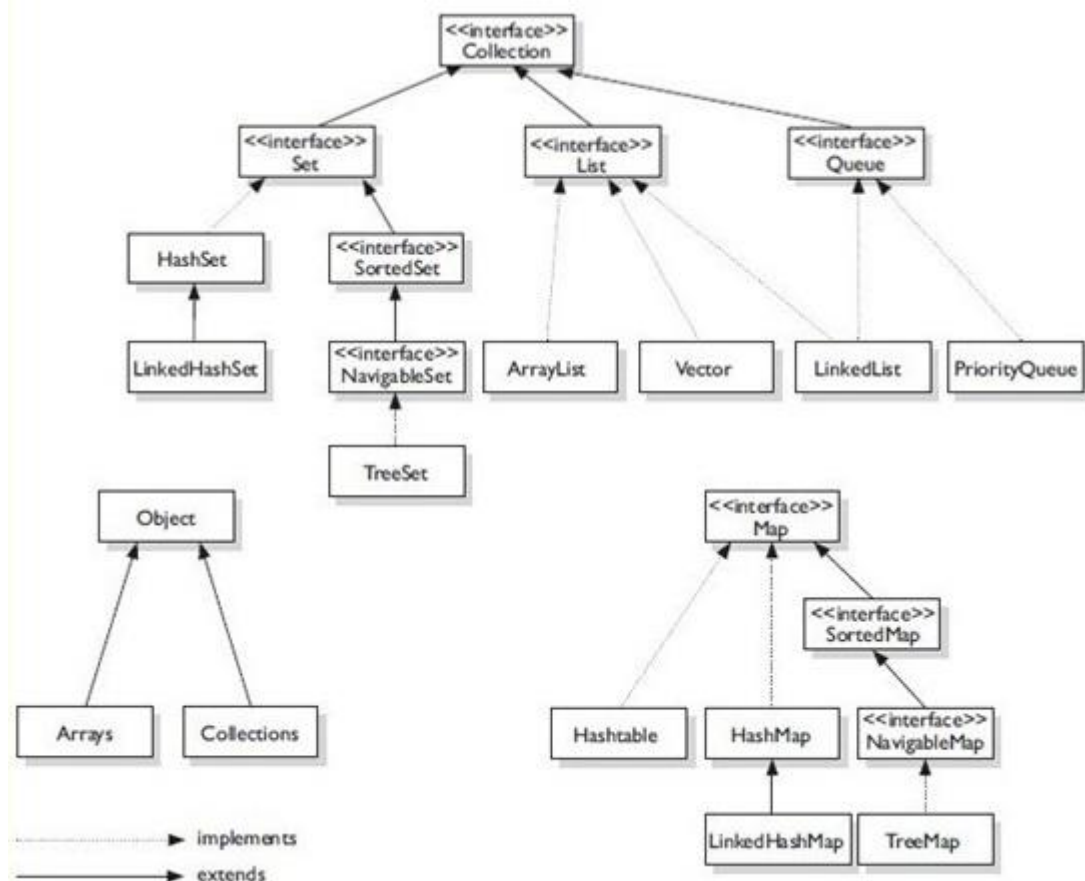
<2> `list`: 有序且可以重复

<3> `map`: 有映射关系的集合(键-值)

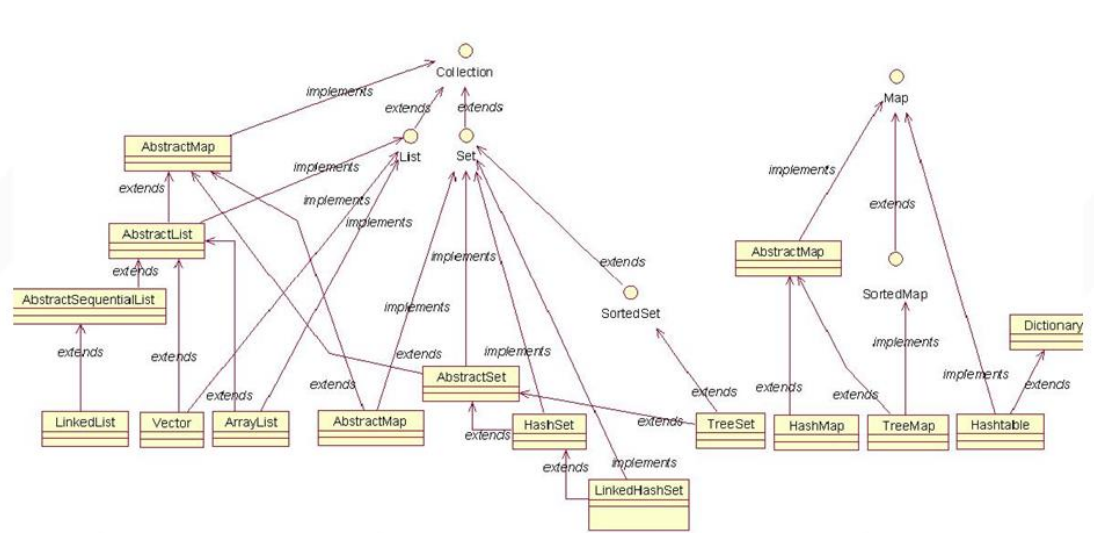
<4> `queue`: 队列集合







add(E e)	将指定对象添加到集合中
remove(Object o)	将指定的对象从集合中移除，移除成功返回 true,不成功返回 false
contains(Object o)	查看该集合中是否包含指定的对象，包含返回 true,不包含返回 false
size()	返回集合中存放的对象的个数。返回值为 int
clear()	移除该集合中的所有对象，清空该集合。
iterator()	返回一个包含所有对象的 iterator 对象，用来循环遍历
toArray()	返回一个包含所有对象的数组,类型是 Object
toArray(T[] t)	返回一个包含所有对象的指定类型的数组



	有序否	允许元素重复否
Collection	否	是
List	是	是
Set	AbstractSet	否
	HashSet	
	TreeSet	
Map	AbstractMap	使用key-value来映射和存储数据，Key必须惟一，value可以重复
	HashMap	
	TreeMap	

2、Set接口

Set关心唯一性，它不允许重复。

HashSet 当不希望集合中有重复值，并且不关心元素之间的顺序时可以使用此类。

LinkedHashSet 当不希望集合中有重复值，并且希望按照元素的插入顺序进行迭代遍历时可采用此类。

TreeSet 当不希望集合中有重复值，并且希望按照元素的自然顺序进行排序时可采用此类。（自然顺序意思是某种和插入顺序无关，而是和元素本身的内容和特质有关的排序方式，譬如“abc”排在“abd”前面。）

3、Queue接口

Queue用于保存将要执行的任务列表。

LinkedList 同样实现了Queue接口，可以实现先进先出的队列。

PriorityQueue 用来创建自然排序的优先级队列。番外篇中有个例子

<http://android.yaohuji.com/archives/3454>你可以看一下。

4、Map接口

Map关心的是唯一的标识符。他将唯一的键映射到某个元素。当然键和值都是对象。

HashMap 当需要键值对表示，又不关心顺序时可采用HashMap。

Hashtable 注意Hashtable中的t是小写的，它是HashMap的线程安全版本，现在已经很少使用。

LinkedHashMap 当需要键值对，并且关心插入顺序时可采用它。

TreeMap 当需要键值对，并关心元素的自然排序时可采用它。

1. Q: ArrayList和Vector有什么区别? HashMap和HashTable有什么区别?

A: Vector和HashTable是线程同步的(synchronized)。性能上, ArrayList和HashMap分别比Vector和Hashtable要好。

2. Q: 大致讲解java集合的体系结构

A: List、Set、Map是这个集合体系中最主要的三个接口。

其中List和Set继承自Collection接口。

Set不允许元素重复。HashSet和TreeSet是两个主要的实现类。

List有序且允许元素重复。ArrayList、LinkedList和Vector是三个主要的实现类。

Map也属于集合系统, 但和Collection接口不同。Map是key对value的映射集合, 其中key列就是一个集合。key不能重复, 但是value可以重复。HashMap、TreeMap和Hashtable是三个主要的实现类。

SortedSet和SortedMap接口对元素按指定规则排序, SortedMap是对key列进行排序。

3. Q: Comparable和Comparator区别

A: 调用java.util.Collections.sort(List list)方法来进行排序的时候, List内的Object都必须实现了Comparable接口。

java.util.Collections.sort(List list, Comparator c), 可以临时声明一个Comparator 来实现排序。

```
Collections.sort(imageList, new Comparator() {  
    public int compare(Object a, Object b) {  
        int orderA = Integer.parseInt( (Image) a).getSequence();  
        int orderB = Integer.parseInt( (Image) b).getSequence();  
        return orderA - orderB;  
    }  
});
```

如果需要改变排列顺序

改成return orderb - orderA 即可。

Set: Set 接口 不允许包含相同的元素 方法用 equal()来比较 返回 true, 则 set 不接受这两个对象

HashSet 是 set 接口的典型实现, HashSet 按 hash 算法来存储集合中的元素 具有很好存储和查找性能 比较方法 equal() 和 hashCode() 允许有 null 值

List: Arraylist 和 vector 是 list 接口的两个典型实现,区别: vector 是线性安全的 性能比 Arraylist 要低 相同: 基于数组实现的 list 类。List 还有一个基于链表实现的 LinkedList 类,插入和删除的速度非常快 即实现了 List 接口, 也实现了 Dueue 接口(双向队列)。也可以用栈使用 List 就相当于所有 key 都是 int 型的 Map

Quque: 用于模拟队列的数据结构。LinkedList 和 ArrayDueue 是其两个比较常用的实现类。

Map: 常用的实现类: HashMap、HashTable、TreeMap 。TreeMap 是基于红黑树对 TreeMap 中所有 key 进行排序, 是有序的。 HashMap 和 HashTable 区别:
<1>HashTable 线程安全 性能差些<2>HashMap 的 key 或者 value 可以为 null Map 中的 value 可以重复 <3>数据增长:当需要增长时,Vector 默认增长为原来一倍,而 ArrayList 却是原来的一半

集合类还提供了一个工具类 Collections。主要用于查找、替换、同步控制、设置不可变集合

Arraylist 和 LinkedList 区别: Arraylist 是顺序存储的线性表 采用数组实现 LinkedList 是链式存储(双向链表)。随机存储比较频繁的元素操作 Arraylist 是可变数组 读取效率很高 排序使用 Collections 类的 sort() 经常需要增加、删除元素应该选用 LinkedList。 Arraylist 比 LinkedList 性能好 Arraylist 线程不安全

LinkedList:下标从0开始 注:若数组没有添加下标为0的元素 `list6.add(0, "123");`

则 `list6.addFirst("asasa");` 默认下标为 0

数组和 List 的转换:

```
String[] sa = {"one", "two", "three", "four"};
List list = Arrays.asList(sa);
```

Arrays 类

四个基本方法:

1. 比较两个数组是否相等的 `equals()`
2. 填充的 `fill()`
3. 对数组进行排序的 `sort()`

```
Collections.sort(list);
```

```
Collections.sort(listPerson1, Collections.reverseOrder());
```

```
Collections.reverse(listPerson1);
```

4. 在一个已排序的数组中查找元素的 `binarySearch()`

赋值数组: `System.arraycopy()` 对象数组和 `primitive` 数组都能拷贝 对象数组只是拷贝的他的 `reference` (引用) 浅拷贝

百度腾讯

1. 全排列

```
#include <iostream>
```

```
using namespace std;
```

```
void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

```
void permutate(int a[], int n, int j) {
    if (j == n-1) {
        for (int i = 0; i < n; ++i) {
            cout << a[i] << " ";
        }
        cout << endl;
    } else {
```

```

        for (int i = j; i < n; ++i) {
            swap(a, i, j);
            permutate(a, n, j+1);
            swap(a, i, j);
        }
    }
}

int main() {
    int a[] = {1, 2, 3, 4};
    permutate(a, 4, 0);
    return 0;
}

```

2. 给定一个数组 `int a[] = {4, 5, 4, 3, 2, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 9}`; 数组相邻元素的差的绝对值为 1，输入数组 a 和待查找整数 t，求 t 在数组 a 中位置。

```

int search(int a[], int n, int t) {
    int i = 0;
    while (i < n && a[i] != t) {
        i += abs(t - a[i]);
    }
    if (i < n) {
        return i;
    }
    return -1;
}

```

3. 一棵二叉树的高度为最深的叶子节点到跟节点的路径长度，此二叉树的宽度为某层节点数最大值，求此二叉树的面积。

```

#include <iostream>
#include "math.h"
#include <queue>
#include "string.h"
using namespace std;

```

```

struct TreeNode {
    int data;
    TreeNode *left;
    TreeNode *right;
    int level;
};

```

```

void createTree(TreeNode **root) {
    *root = new TreeNode();
}

```

```

(*root)->left = new TreeNode();
(*root)->right = new TreeNode();
(*root)->left->left = new TreeNode();
(*root)->left->right = new TreeNode();
(*root)->left->left->left = new TreeNode();
(*root)->left->left->right = new TreeNode();
(*root)->left->left->left->left = new TreeNode();
(*root)->left->left->left->left->left = new TreeNode();
(*root)->right->right = new TreeNode();
}

```

```

void clearTree(TreeNode *root) {
    if (root != NULL) {
        clearTree(root->left);
        clearTree(root->right);
        cout << "clear " << &root << endl;
        delete root;
    }
}

```

```

int height(const TreeNode *root) {
    if (root != NULL) {
        int left = height(root->left);
        int right = height(root->right);
        return (left > right ? left : right) + 1;
    }
    return 0;
}

```

```

int Area(TreeNode *root) {
    int depth = height(root); //计算二叉树的层数
    cout << "深度" << depth << endl;
    int *count = new int[depth]; //每一层都一个计数器
    memset(count, 0, depth*sizeof(count));
    root->level = 0;
    queue<TreeNode*> Q;
    Q.push(root);
    while (!Q.empty()) {
        TreeNode *cur = Q.front(); Q.pop();
        count[cur->level]++;
        if (cur->left != NULL) {
            cur->left->level = cur->level + 1;
            Q.push(cur->left);
        }
    }
}

```

```

        if (cur->right != NULL) {
            cur->right->level = cur->level + 1;
            Q.push(cur->right);
        }
    }
    int width = 0;
    for (int i = 0; i < depth; ++i) {
        if (count[i] > width) {
            width = count[i];
        }
    }
    cout << "宽度" << width << endl;
    return depth * width;
}

int main() {
    TreeNode *root;
    createTree(&root);
    Area(root);
    clearTree(root);
    return 0;
}

//精简版
int getArea(TreeNode *root) {
    if (!root) return 0;
    queue<TreeNode*> Q;
    Q.push(root);
    int last = 1;
    int current = 1;
    int width = 1;
    int height = 0;
    while(!Q.empty()) {
        int t = last;
        while (t-- > 0) {
            TreeNode* p = Q.front(); Q.pop();
            if (p->left) Q.push(p->left);
            if (p->right) Q.push(p->right);
        }
        current = Q.size();
        width = current > width ? current : width;
        last = current;
        height++;
    }
}

```

```
    return height * width;
}
```

4. TCP 和 UDP 的区别

TCP 是一种面向连接的、可靠的、基于字节流的传输层协议，具有**差错检测**、流量控制和拥塞控制功能（error detection、flow control、congestion control）。基于 TCP 的上层协议有 HTTP、TLS、POP3（客户端与邮件服务器）、SMTP（邮件服务器之间）、FTP 等。

UDP 是一种面向非连接的、不可靠的、基于数据报的传输层协议。基于 UDP 的上层协议有 DNS（域名解析）、RTP（多媒体，看视频）、SNMP。

HTTP 的特点？

TLS 安全机制？

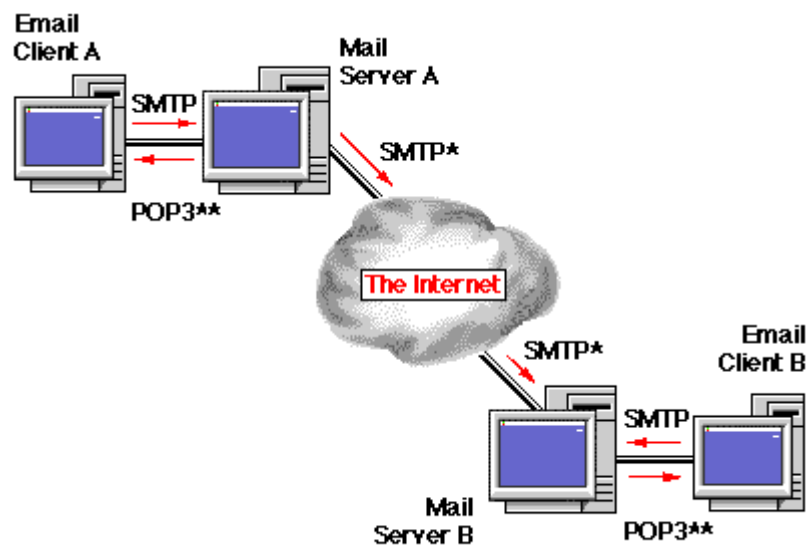
FTP 的工作原理？

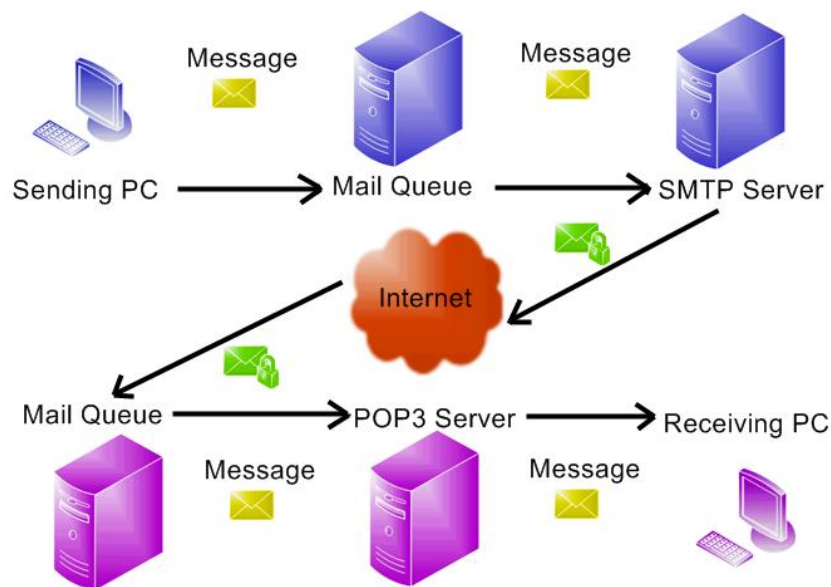
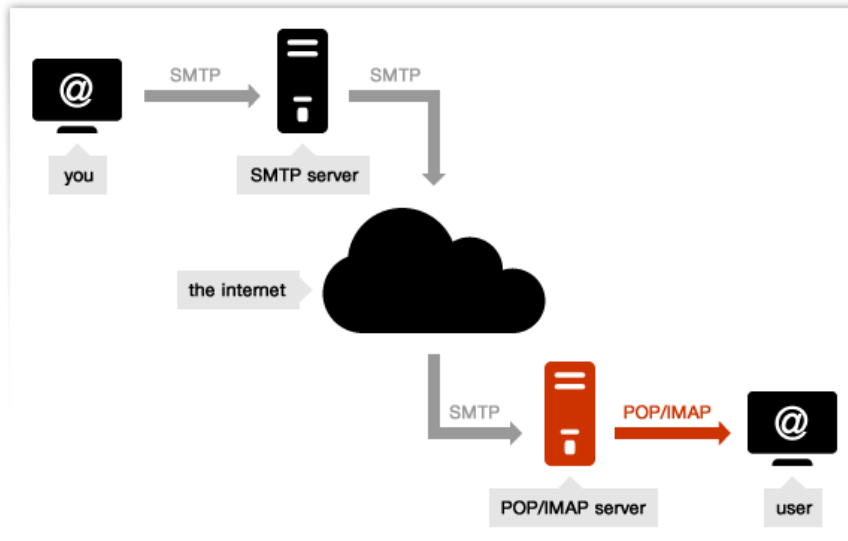
DNS 的工作原理？

SNMP 的用途？

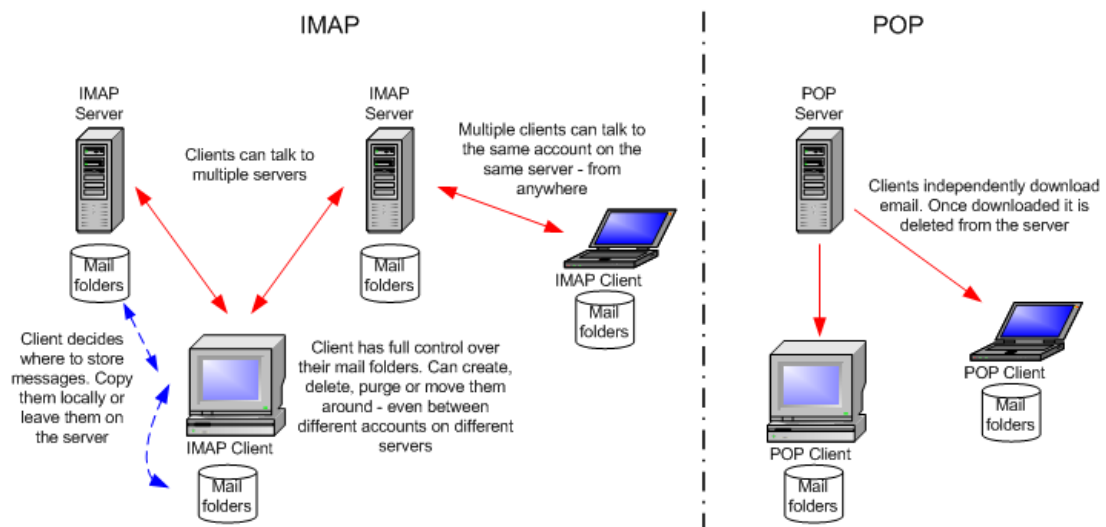
4.1 POP3 与 SMTP 的区别？

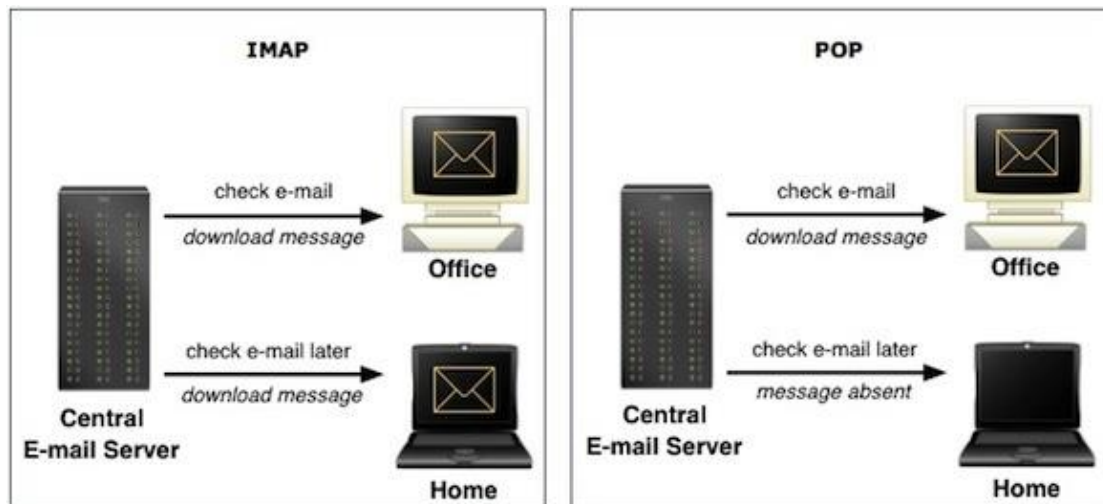
POP is a protocol for storage of email. SMTP is a protocol for sending and receiving email.





POP3 与 SMTP 数据流程图

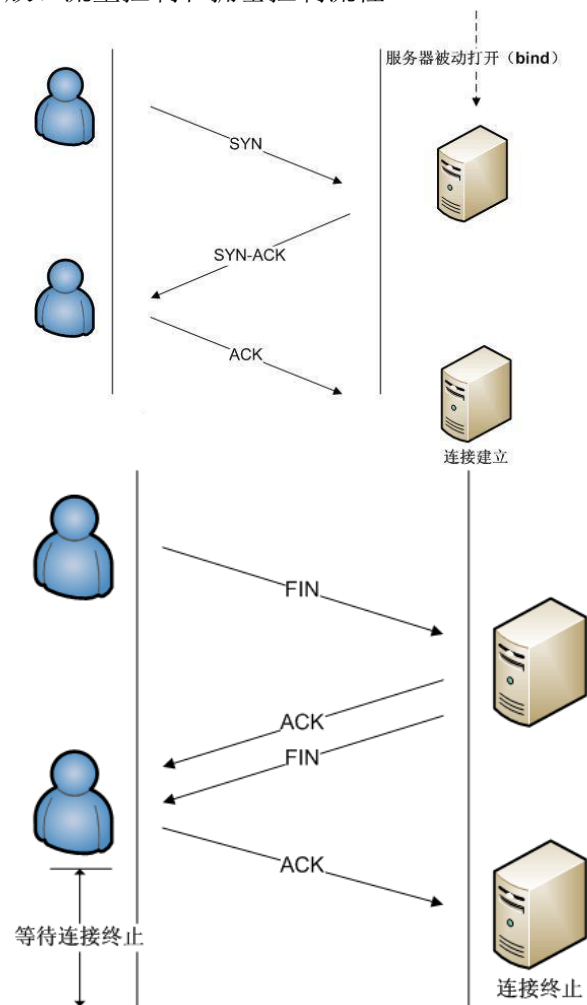




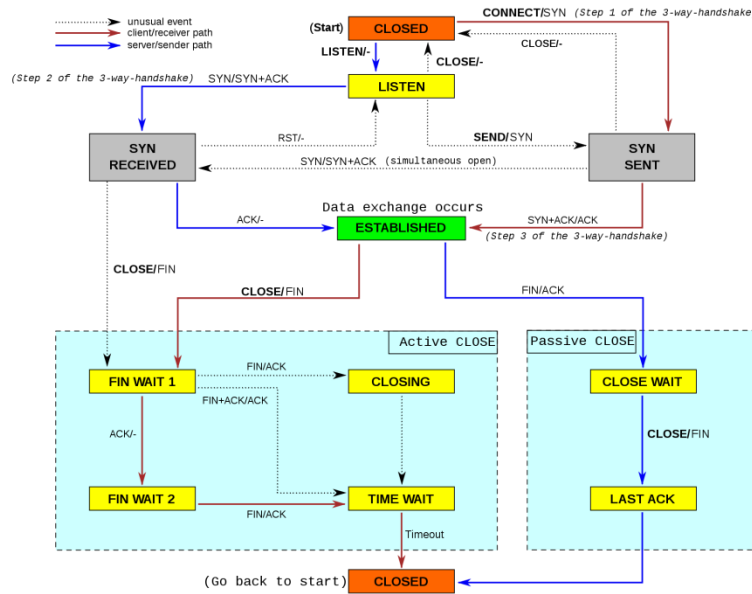
the difference of POP3 and IMTP(Internet Mail Access Protocol)

从今天开始，习惯使用图片搜索，深刻体会图片表达的强大！

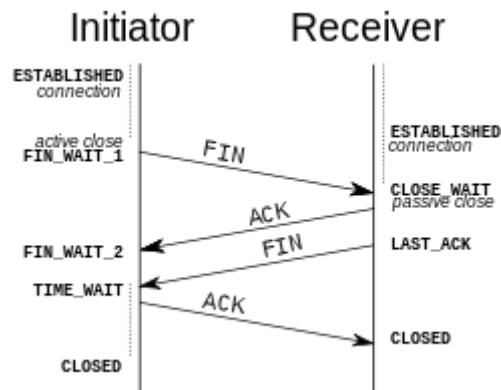
4.2 TCP 的建立、释放、流量控制和拥塞控制流程？



TCP 连接的建立和释放



TCP 状态机



5. 可重入函数

在 `read`、`time`、`localtime` 和 `new` 函数中，只有 `new` 函数是可重入的。

若一个函数是可重入的，则该函数：

- 不能含有静态（全局）非常量数据。
- 不能返回静态（全局）非常量数据的地址。
- 只能处理由调用者提供的数据。
- 不能依赖于单实例模式资源的锁。
- 不能调用(call)不可重入的函数(有调用到的函数需满足前述条件)。

多“用户/对象/进程优先级”以及多进程，一般会使得对可重入代码的控制变得复杂。同时，IO 代码通常不是可重入的，因为他们依赖于像磁盘这样共享的、单独的(类似编程中的静态(Static)、全域(Global)资源)。

6. 判断两个矩形是否相交

假定矩形是用一对点表达的(minx, miny) (maxx, maxy)，那么两个矩形

`rect1{(minx1, miny1)(maxx1, maxy1)}`


```
rect2{(minx2, miny2)(maxx2, maxy2)}
```

相交的结果一定是个矩形,构成这个相交矩形 rect{(minx, miny) (maxx, maxy)}的点对坐标是:

```
minx   =   max(minx1,   minx2)
miny   =   max(miny1,   miny2)
maxx   =   min(maxx1,   maxx2)
maxy   =   min(maxy1,   maxy2)
```

如果两个矩形不相交,那么计算得到的点对坐标必然满足:

(minx > maxx) 或者 (miny > maxy)

判定是否相交,以及相交矩形是什么都可以用这个方法一体计算完成。

从这个算法的结果上,我们还可以简单的生成出下面的两个内容:

(一) 相交矩形: (minx, miny) (maxx, maxy)

(二) 面积: 面积的计算可以和判定一起进行

```
if ( minx>maxx ) return 0;
if ( miny>maxy ) return 0;
return (maxx-minx)*(maxy-miny)
```

参考链接: <http://www.cnblogs.com/0001/archive/2010/05/04/1726905.html>

7. 求一棵二叉树任意两个节点的最近公共祖先

```
#include <iostream>
#include "math.h"
#include <queue>
#include "string.h"
using namespace std;
```

```
struct TreeNode {
    int data;
    TreeNode *left;
    TreeNode *right;
    int level;
    TreeNode(int data = 0, TreeNode *left = NULL, TreeNode *right = NULL, int
level = 0) {
        this->data = data;
        this->left = left;
        this->right = right;
        this->level = level;
    }
};
```

```
void createTree(TreeNode **root) {
```

```

    *root = new TreeNode(1);
    (*root)->left = new TreeNode(2);
    (*root)->right = new TreeNode(3);
    (*root)->left->left = new TreeNode(4);
    (*root)->left->right = new TreeNode(5);
    (*root)->left->left->left = new TreeNode(6);
    (*root)->left->left->right = new TreeNode(7);
    (*root)->left->left->left->left = new TreeNode(8);
    (*root)->left->left->left->left->left = new TreeNode(9);
    (*root)->right->right = new TreeNode(10);
}

void clearTree(TreeNode *root) {
    if (root != NULL) {
        clearTree(root->left);
        clearTree(root->right);
        cout << "clear " << &root << endl;
        delete root;
    }
}

int findLCA(TreeNode *root, TreeNode *a, TreeNode *b, TreeNode **lca) {
    //根节点为空，自不必说，终止递归查找
    //当*lca不为空，说明已经找到最近公共祖先，可进行剪枝
    if (root == NULL || *lca != NULL) {
        return 0;
    }
    int flag = 0, nLen = 0;
    if (root == a || root == b) {
        flag = 1;
    }
    nLen += flag;
    nLen += findLCA(root->left, a, b, lca);
    nLen += findLCA(root->right, a, b, lca);
    if (nLen == 2 && flag == 0 && *lca == NULL) {
        *lca = root;
    }
    return nLen;
}

int main() {
    TreeNode *root;
    createTree(&root);
}

```

```

TreeNode *a = root->left->left;
TreeNode *b = root->left->left->left->left;

printf(" a->data = %d\n", a->data);
printf(" b->data = %d\n", b->data);

TreeNode *out = NULL;
int ret = findLCA(root, a, b, &out);
if (ret == 2) {
    printf("the least common ancestor = %p, %d\n", out, out->data);
} else {
    printf("no LCA!\n");
}
clearTree(root);
return 0;
}

```

算法时间复杂度 $f(n) = \sum_{k=1}^h (2^k \cdot 2^{h-k} + 2^{k-1}) = O(n \log_2 n)$

参考网址:

http://blog.csdn.net/piaojun_pj/article/details/5971125?reload#reply

http://blog.csdn.net/v_july_v/article/details/11921021

8. int *&p 和 int*&p 的区别

int *&p 指向引用的指针是非法的;

int *&p, 实际上可以这么看 int* &p, 即&p 表示一个引用变量, 其引用变量的类型为*int。

1. 在线笔试题回顾

附近的商户

逛了一天, 掏出点评客户端, 选一个最近的商户大吃一顿! 请设计一套快速搜索用户附近的商户的算法。

请注意: 先写下您的思路, 然后提供代码或伪代码。

问题:

(10分)

输入一个整数数组, 调整数组中数字的顺序, 使得所有奇数排在所有偶数前面。

请注意: 先写下您的思路, 然后提供代码或伪代码。

2. 一面问题回顾

问项目

问 Java 的泛型和 C++ 的模板的区别

问 Java final 关键字的作用?

final 类不能被继承、final 方法不能被覆盖 (优点: 阻止子类修改它的意义和实现; 编译器遇

到 **final** 方法时会转入内嵌机制，大大提高执行效率)、**final** 修饰变量表示常量，一旦初始化就不能改变。

问 Java 线程锁

问一个概率题 重男轻女的村子

假设一个村子里，村民们都有重男轻女的观念，每一对夫妻都会不断地生产，直到生出一个男孩。如果生男生女的概率是一样的，那这个村子最终的男女比例应该是多少？

解答： 如果这个村子里有 C 对夫妻，那最后就有 C 个男孩。而女孩的数量呢？这是一个数学上求期望值的问题。

不妨任选一户人家来分析，设这户人家有 n 个女孩。如果 $n=0$ ，也就是说这户人家第一次就生了个男孩，那么这个概率便是 $1/2$ 。 $n=1$ 时，便是先女后男，概率是 $1/2 * 1/2 = 1/4$ 。 $n=2$ 时，便是前两次生了两个女孩，第三次生了男孩，概率是 $1/2 * 1/2 * 1/2 = 1/8$ 。以此类推，一户人家出现 n 个女孩的概率是 $(1/2)^{n+1}$ 。由此可以算出一户人家女孩数量的期望值是 $0 + 1/4 + 2/8 + 3/16 + \dots = 1$ 。因此， C 户人家的女孩数量的期望值便是 C ，女孩和男孩的数量在期望上是相等的。也就是说，即使是在一个重女轻男的地方，男女比例仍然是 $1:1$ 。

2.1 单链表反转

2.2 求单链表中间的节点

2.3 非递归中序遍历二叉树

淘米

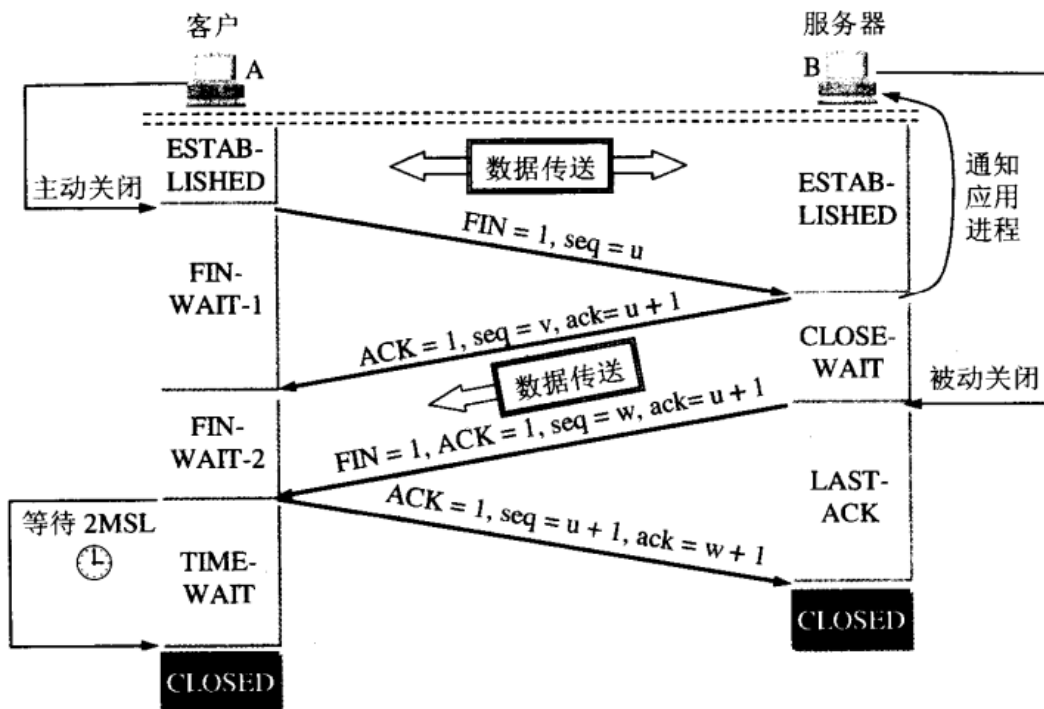
一、单项选择题

1. 一种既有利于短小作业又兼顾到长作业的作业调度算法是 ()。

选项:

- a、先来先服务
- b、轮转
- c、最高响应比优先
- d、均衡调度

2. TCP 的连接释放的四次握手问题



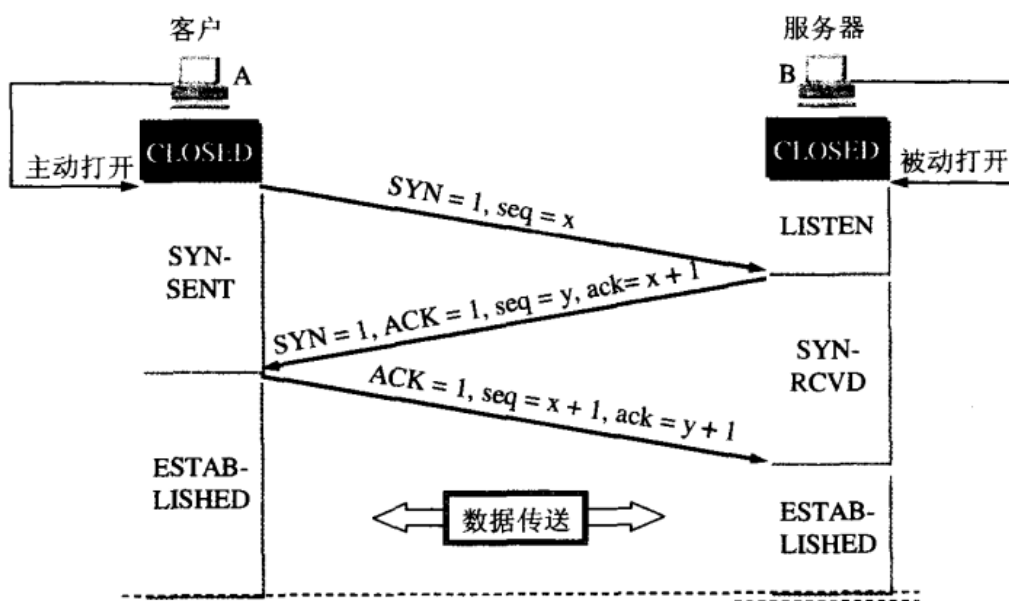
当客户端 A 向客户端 B 发送 $FIN = 1, seq = u$ ，客户端 B 响应什么？ $ACK = 1, ack = u + 1$ 。

当客户端 A 向客户端 B 发送 FIN ，客户端 B 也已回发响应，此时客户端 B 处于什么状态？

CLOSE_WAIT 状态。（留意客户端 A 和客户端 B 的状态）

客户端 A 发送 FIN ，收到客户端 B 的响应，并且客户端 B 也发送 FIN ，客户端 A 也发送响应后，客户端 A 进入 TIME_WAIT 状态，等待时间为 2MSL，MSL 即为 Maximum Segment Length，最大报文长度，RFC 793 建议为 2 分钟，2MSL=4 分钟。

顺便复习 TCP 链接建立的三次握手



二. 应用题

1. 过河问题

一个男人一个女人，两个男孩两个女孩，一个猎人一只狗，他们要过河，只有一条船，能载两个人过河，只有男人、女人、猎人会划船，男人不在女人打男孩，女人不在男人打女孩，猎人不在狗咬所有人，怎么让他们安全过河？

- 答：1) 猎人、狗过河，猎人回；
2) 猎人、男孩 1 过河，猎人、狗回；
3) 男人、男孩 2 过河，男人回；
4) 男人、女人过河，女人回；
5) 猎人、狗过河，男人回；
6) 男人、女人过河，女人回；
7) 女人、女孩 1 过河，猎人、狗回；
8) 猎人、女孩 2 过河，猎人回；
9) 猎人、狗过河，结束。

类似的渊源问题：

一个猎人带着一只羊，一只狼和一棵白菜回家，路上遇到一条河。河边只有一条船。但船太小，一次最多只可载猎人和另一样东西过河，但猎人不在时，狼要吃羊，羊要吃白菜。请问怎样才能把狼，羊，白菜都安全地过河？

- 答：1) 猎人、羊过河，猎人回；
2) 猎人、狼过河，猎人、羊回；
3) 猎人、白菜过河，猎人回；
4) 猎人、羊过河，结束。

三、算法题

将一个有序序列的中间的数，称为该序列的中位数。当该序列长度为奇数时，其中位数即为当中的数，如序列{1,3,21,100,1000}中位数为 21；当该序列长度为偶数时，其中位数为当中两个数的平均值，如序列{1,3,13,21,100,1000}中位数为 $(13+21)/2=22$ 。求一个二叉树所有节点值的中位数。

二叉搜索树的中位数

<http://blog.csdn.net/hhygcy/article/details/4654305>

二叉搜索树的中序遍历的序列正好是有序的。

二叉搜索树转双向链表->再求中位数

二叉树转双向链表

```
void bst2dll(TreeNode* root, TreeNoe* &tail) {
    if (root) {
        bst2dll(root->left);
        root->left = tail; //已转化为双向链表部分的尾指针
        if (tail) {
            tail->right = root;
        }
        tail = root;
        bst2dll(root->right);
    }
}
```

```

}
int medianInBST(TreeNode* root) {
    TreeNode* tail = NULL;
    bst2dll(root, tail);
    if (!tail) return -1;
    TreeNode* fast = tail;
    TreeNode* slow = tail;
    while (fast && slow) {
        if (!fast->left) {
            return slow->data;
        } else if (fast->left && !fast->left->left) {
            return (slow->data + slow->left->data) >> 1;
        } else {
            fast = fast->left->left;
            slow = slow->left;
        }
    }
}
}

```

快慢指针问题

1. 将循环单链表的环去掉

```

bool hashCicle(Node* head, Node* &encounter) {
    Node* fast = head, *slow = head;
    while (fast && fast->next) {
        fast = fast->next->next;
        slow = slow->next;
        if (fast == slow) {
            encounter = fast;
            return true;
        }
    }
    encounter = NULL;
    return false;
}

```

从起点到入口点距离 x ，从入口点到相遇点距离为 y ，

根据慢指针移动的距离得到

$$x + y = s$$

根据快指针移动的距离得到

$s + nr = 2s$ (慢指针走 s 步，相遇后，快指针比慢指针多走 nr 步，又快指针走的步数是慢指针的 2 倍)

则有 $x + y = nr$

$$\text{即 } x = nr - y = (n-1)r + r - y \quad (n \geq 1)$$

这样就得出一个结论：让指针 $p1$ 从起点开始遍历，指针 $p2$ 从 $encounter$ 开始遍历，且步长均为 1，当 $p1$ 移动 x 步后到达入口点，此时 $p2$ 也移动 x 步 ($= (n-1)r + r - y$ 步)， $p2$ 也到达入

口点。

```
Node *findEntry(Node* head, Node *encounter) {
    Node* p1 = head;
    Node* p2 = encounter;
    while (p1 != p2) {
        p1 = p1->next;
        p2 = p2->next;
    }
    return p1;
}
```

参考: <http://hi.baidu.com/iwitggwg/item/7ad684119a27fefc9c778a5c>

复习和二叉树相关的面试题

1. 二叉树的遍历, 3 种递归遍历以及非递归遍历, 层次遍历

二叉树非递归后续遍历

```
void postOrder(TreeNode* root) {
    stack<TreeNode*> S;
    TreeNode* cur = NULL;
    TreeNode* pre = NULL;
    S.push(root);
    while (!S.empty()) {
        cur = S.top();
        if ((cur->left == NULL && cur->right == NULL) ||
            (pre != NULL && (pre == cur->left || pre == cur->right))) {
            cout << cur->data << " ";
            pre = cur;
            S.pop();
        } else {
            if (cur->left) S.push(cur->left);
            if (cur->right) S.push(cur->right);
        }
    }
}
```

2. 求二叉树的[两个任意节点的最近公共祖先](#)

3. 如何判断一个二叉树是否是平衡的?

4. 在二叉树中找出和为某一值的所有路径?

5. 判断整数序列是不是二叉搜索树的后序遍历结果?

解题思路: 在标识出左右子树时, 遍历右子树, 发现右子树中的元素小于其父节点, 则该序列为错误的。

```
bool checkIsLastSuq(int* data, int start, int end) {
    if(start >= end) return true;
    int mid = 0;
    for(mid = start; mid < end && data[mid] < data[end]; ++mid) {
```



```

        ;
    }
    //mid是第一个不小于data[end]的元素的下标
    for (int i = mid; i < end; i++) {
        if (data[i] < data[end]) {
            return false;
        }
    }
    bool bLeft = checkIsLastSuq(data, start, mid - 1);
    bool bRight = checkIsLastSuq(data, mid + 1, end - 1);
    return bLeft && bRight;
}

```

参考: <http://blog.csdn.net/cscmaker/article/details/8589489>

6. 通过先序遍历和中序遍历的序列构建二叉树。

```

int search(char arr[], int start, int end, char value) {
    for (int i = start; i <= end; i++) {
        if (arr[i] == value) return i;
    }
}

TreeNode* buildTree(char in[], char pre[], int start, int end) {
    static int preIndex = 0;
    if (start > end) return NULL;
    TreeNode* p = newNode(pre[preIndex++]);
    if (start == end) return p;
    int inIndex = search(in, start, end, p->data);
    p->left = buildTree(in, pre, start, inIndex - 1);
    p->right = buildTree(in, pre, inIndex + 1, end);
    return p;
}

```

7. 将一个有序的整数数组放到二叉搜索树中

```

void arrayToTree(int a[], int begin, int end, TreeNode* &root) {
    if (begin <= end) {
        int mid = (begin + end) / 2;
        root = new TreeNode();
        root->data = a[mid];
        root->left = NULL;
        root->right = NULL;
        arrayToTree(a, begin, mid - 1, root->left);
        arrayToTree(a, mid + 1, end, root->right);
    }
}

```

8. 将一棵满二叉树中同一层的节点的兄弟节点指向其右边的节点。注意：是一棵满二叉树。

```
void link_sibling(Node* root) {
    if (!root) return;
    root->sibling = NULL;
    for(Node* p = root; p->left; p = p->left) {
        Node* up = p;
        Node* cur = up->left;
        while (cur) {
            if (cur == up->left) {
                cur->sibling = up->right;
            } else {
                up = up->sibling;
                if (up) {
                    cur->sibling = up->left;
                } else {
                    cur->sibling = NULL;
                }
            }
            cur = cur->sibling;
        }
    }
}
```

```
void visit_sibling(Node* root) {
    for (Node* p = root; p; p = p->left) {
        for( Node* q = p; q; q = q->sibling) {
            cout << q->data << " ";
        }
        cout << endl;
    }
}
```

9. 比较两个二叉树是否相等，分为左右子树是否重要两种情况。

```
bool equals(TreeNode* a, TreeNode* b) {
    if (!a && !b) return true;
    else if ( (a && !b) || (!a && b) || a->data != b->data) return false;
    else {
        bool left = equals(a->left, b->left);
        if (!left) return false;
        bool right = equals(a->right, b->right);
        return right;
    }
}
```

10. 求二叉树的面积

```
int getArea(TreeNode *root) {
```

```

    if (!root) return 0;
    queue<TreeNode*> Q;
    Q.push(root);
    int last = 1;
    int current = 1;
    int width = 1;
    int height = 0;
    while(!Q.empty()) {
        int t = last;
        while (t-- > 0) {
            TreeNode* p = Q.front(); Q.pop();
            if (p->left) Q.push(p->left);
            if (p->right) Q.push(p->right);
        }
        current = Q.size();
        width = current > width ? current : width;
        last = current;
        height++;
    }
    return height * width;
}

```

参考: [求二叉树的深度和宽度](#)

11. 求二叉树中节点间最大距离

```

int getDepth(TreeNode* root) {
    if (!root) return 0;
    int ld = getDepth(root->left);
    int rd = getDepth(root->right);
    return 1 + (ld > rd ? ld : rd);
}

int maxdis = 0;

void findMaxDis(TreeNode* root) {
    if (!root) return;
    int ldis = 0, rdis = 0;
    if (root->left) {
        ldis = getDepth(root->left);
    }
    if (root->right) {
        rdis = getDepth(root->right);
    }
}

```

```

    }
    if ((ldis + rdis) > maxdis) {
        maxdis = ldis + rdis;
    }
    findMaxDis(root->left);
    findMaxDis(root->right);
}

```

参考: <http://blog.csdn.net/cxh342968816/article/details/6656473>

12. 求二叉树最大路径和

看看吧, 后续遍历是这样做的: 左右根, 所以访问的最有一个节点实际上就是整棵二叉树的根节点 **root**: 然后, 找到第一个大于该节点值的根节点 **b**, **b** 就是 **root** 右子树最左边的节点 (大于根节点的最小节点)。那么 **b** 前面的就是 **root** 的左子树。既然是二叉搜索树的遍历结果, 那么在 **b** 和 **root** 之间的遍历结果, 都应该大于 **b**。去拿这个作为判断的条件。

参考: <http://blog.csdn.net/randyjiawenjie/article/details/6772145>

智能指针

实现一个 hash_set

6. 判断两个矩形是否相交

假定矩形是用一对点表达的(minx, miny) (maxx, maxy), 那么两个矩形

rect1{(minx1, miny1)(maxx1, maxy1)}

rect2{(minx2, miny2)(maxx2, maxy2)}

相交的结果一定是个矩形, 构成这个相交矩形 rect{(minx, miny) (maxx, maxy)}的点对坐标是:

minx = max(minx1, minx2)

miny = max(miny1, miny2)

maxx = min(maxx1, maxx2)

maxy = min(maxy1, maxy2)

如果两个矩形不相交, 那么计算得到的点对坐标必然满足:

(minx > maxx) 或者 (miny > maxy)

判定是否相交, 以及相交矩形是什么都可以用这个方法一体计算完成。

从这个算法的结果上, 我们还可以简单的生成出下面的两个内容:

(一) 相交矩形: (minx, miny) (maxx, maxy)

(二) 面积: 面积的计算可以和判定一起进行

```

    if ( minx>maxx ) return 0;
    if ( miny>maxy ) return 0;
    return (maxx-minx)*(maxy-miny)

```

参考链接: <http://www.cnblogs.com/0001/archive/2010/05/04/1726905.html>

```

int findLCA(TreeNode *root, TreeNode *a, TreeNode *b, TreeNode **lca) {
    //根节点为空, 自不必说, 终止递归查找
    //当*lca不为空, 说明已经找到最近公共祖先, 可进行剪枝
    if (root == NULL || *lca != NULL) {
        return 0;
    }
    int flag = 0, nLen = 0;
    if (root == a || root == b) {
        flag = 1;
    }
    nLen += flag;
    nLen += findLCA(root->left, a, b, lca);
    nLen += findLCA(root->right, a, b, lca);
    if (nLen == 2 && flag == 0 && *lca == NULL) {
        *lca = root;
    }
    return nLen;
}

```

约瑟夫问题

n 个人 (编号 $0 \sim (n-1)$), 从 0 开始报数, 报到 $(m-1)$ 的退出, 剩下的人继续从 0 开始报数。求胜利者的编号。

我们知道第一个人(编号一定是 $(m-1) \bmod n$) 出列之后, 剩下的 $n-1$ 个人组成了一个新的约瑟夫环 (以编号为 $k = m \bmod n$ 的人开始):

$k, k+1, k+2, \dots, n-2, n-1, 0, 1, 2, \dots, k-2$

并且从 k 又以 0 开始报数。 注: $k-1$ 已经出局。

现在我们把它们的编号进行一次转换:

$k \rightarrow 0$

$k+1 \rightarrow 1$

$k+2 \rightarrow 2$

...

$k-2 \rightarrow n-2$

转换之后的问题就成为了 $(n-1)$ 个人的报数问题。假设 $(n-1)$ 个人的报数问题的解是 x , 即 x 是最终的胜利者, 把 x 放回又变成 n 个人报数的问题。设 n 个人报数的问题的解为 x' , 根据上述转换有 $x' = (x+k) \bmod n$ 。

同样, 如果知道 $(n-2)$ 个人的报数问题的解, 那么又可以很快地求出 $(n-1)$ 个人的报数问题的解。这显然就是一个递推问题。

令 $x[i]$ 表示 i 个人的报数问题的解, 其中报 $m-1$ 数的人退出, 给出递推公式:

$x[1]=0;$

$x[i]=(x[i-1]+m) \bmod i; (i>1)$

根据递推公式, 从 1-n 顺序算出 $x[i]$ 的值, 则 n 个人的报数问题的解为 $x[n]$ 。
因为在实际生活中, 总是从 1 开始编号, 所以 $x[n]+1$ 即为 n 个人的报数问题的解。

八皇后问题

```
import java.util.Scanner;

public class Main {
    static int[][] queenList = new int[92][8]; //保存八皇后所有摆放位置, 即所有解
    static int[] mark = new int[8];
    static int count = 0;

    public static void main(String[] args) {
        //初始化棋盘和问题的解
        for(int i=0; i<92; i++){
            for(int j=0; j<8; j++){
                queenList[i][j] = -1;
            }
        }
        for(int i=0; i<8; i++){
            mark[i] = -1;
        }

        //求解
        putQueen(0);

        Scanner sysIn = new Scanner(System.in);
        int nTestNumber = sysIn.nextInt();
        while(nTestNumber>0){
            int index = sysIn.nextInt();
            for(int i=0; i<8; i++){

                System.out.print(queenList[index-1][i]);

            }
            System.out.println();
            nTestNumber--;
        }
    }
}
```

```

private static void putQueen(int col) {
    if( col==8 ){
        for(int j=0;j<8;j++){
            queenList[count][j] = mark[j]+1;
        }
        count++;
        return;
    }
    int i=0;
    int j=0;
    //在第 col 列中搜索没有被控制的行号
    for(i=0;i<8;i++){
        for(j=0;j<col;j++){
            if(mark[j]==i ||
Math.abs(j-col)==Math.abs(i-mark[j])){
                break; //同行, 斜向
            }
        }
        if(j==col){
            mark[col]=i; //搜索到了合适的行号
            putQueen(col+1); //进入下一列进行搜索
        }
    }
}
}

```

迷宫问题

```

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

void bfs(int,int);
int end[2]; //终点, 设为全局变量, 减少传参个数
int n; //迷宫规模
char mat[100][100];
bool vis[100][100];
int dir[4][2]={0,1},{1,0},{0,-1},{-1,0}; //向右(x,y+1)、向下
(x+1,y)、向左(x,y-1)、向上(x-1,y)

int main()
{
    int t;
    cin>>t;
}

```

```

int begin[2]; //起点
while(t--)
{
    memset(mat, '#', sizeof(mat));
    memset(vis, false, sizeof(vis));
    cin>>n; //输入迷宫规模
    for(int i=0; i<n; i++)
    {
        cin>>mat[i];
    }
    cin>>begin[0]>>begin[1]>>end[0]>>end[1];

    if(mat[begin[0]][begin[1]]=='#' || mat[end[0]][end[1]]=='#')
//先判断起始点是否合法
    {
        cout<<"NO"<<endl;
        continue;
    }

    bfs(begin[0], begin[1]);
    if(vis[end[0]][end[1]]==true)
        cout<<"YES\n";
    else
        cout<<"NO\n";
}

//四个方向循环，深度优先搜索
void bfs(int x, int y)
{
    //传入的点必须为'.', 否则不予处理
    if(mat[x][y]!='.')
    {
        //处理当前点
        vis[x][y]=true;
        //处理下一阶段的点
        {
            int u, v;
            for(int i=0; i<4; i++) //遍历四个方向
            {
                u=x+dir[i][0];
                v=y+dir[i][1];
                //下标合法性检验
                if(u<0 || u>=n || v<0 || v>=n) continue;
                if(mat[u][v]!='.')

```



```

        {
            if(!vis[u][v])
                bfs(u,v);
        }
    } //end 处理下一阶段的点 else
}
}

```

[LeetCode 124] - 二叉树最大路径和(Binary Tree Maximum Path Sum)

问题

给出一个二叉树，找到其中的最大路径和。

路径可以从树中任意一个节点开始和结束。

例如：

给出如下二叉树，

```

    1
   /\
  2  3

```

返回 6。

初始思路

为了简化分析，我们先假设二叉树中所有节点的值都是正数。通过观察可以发现，一棵二叉树的最大路径，就是其左子树的最大路径加上右子树的最大路径。看起来可以从根节点出发通过深度优先递归来求解：

函数 查找路径

如果是叶子节点，返回叶子节点的值

如果不是叶子节点

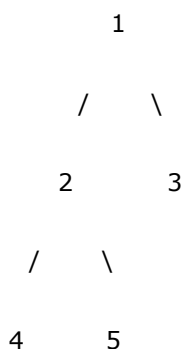
左子树路径和 = 查找路径（左子树）

右子树路径和 = 查找路径（右子树）

如果左子树路径+右子树路径和+当前节点值 > 当前最大路径，更新最大路径

返回左子树路径+右子树路径和+当前节点值

用题目中的简单例子来验证，是可以得出答案的。但是使用复杂一点的树来验证后就发现其中的问题了，如



使用前面的伪代码得出的结果是 15，但是其实答案应该是 11，由 3，1，2，5 或者 2，4，5 得到。分析可以发现问题在于计算 2，4，5 这棵树时，它的最长路径为 11，这是正确的。但是当它作为左子树向父节点返回最长路径时，因该返回 7 而不是 11。因为从 1 出发不走重复路径不可能同时到达 4 或 5 的。通常二叉树节点路径的定义是每个节点只能访问一次，通过测试数据也可以验证题目就是这样要求的。因此我们需要两个最大值，一个是当前树的最大路径，即前面伪代码算出来的那个值；另一个是当前树向父节点提供的最大路径，这个值应该是根节点的值加上路径最长的子树那边的最大路径。我们向上层递归函数返回这个值。

好了，现在全是正数的情况解决了。让我们开始把负数引入。负数引入后，将会导致以下几个变化：

- 叶子节点的值也有可能成为最大路径。在全是正数的情形下，叶子节点的值肯定不可能是最大路径，因为加上父节点的值后必然会变大。有了负数以后，这个情况就不成立了，如：



这时最大路径就是 3。

- 当前树最大路径的计算方法。有了负数以后不能简单的把左子树返回的值，右子树返回的值及当前的值相加了。这里我们把各种情况列举出来：
 - 当前值为正，子树返回值都为正：全相加

- 当前值为正，子树返回值有一个为正：当前值+正的那个值，因为负值只会让结果变小。
- 当前值为正，子树返回值都是负：只取当前值，负值越加越小。
- 当前值为负，子树返回的值都为正：全相加，虽然值会变小，但是没有当前节点左右就不能联通。
- 当前值为负，子树返回值有一个为正：当前值+正的那个值。
- 当前值为负，子树返回值都为负：当前值，负值越加越小。
- 向父节点提供的最大路径的计算方法。和当前树最大路径计算方法基本一样。就是仍然要左子树右子树的值只能取大的那个。

将上面分析转换成代码，并加入一些细节如没有左（右）子树的判断。请注意由于节点的取值范围并没有限定，所以不能使用某个特殊值作为没有左（右）子树的标志。结果如下：

```

1 class Solution {
2 public:
3     int maxPathSum(TreeNode *root)
4     {
5         if(!root)
6         {
7             return 0;
8         }
9
10        maxSum_ = 0;
11        firstValue_ = true;
12
13        CountPathSum(root);
14
15        return maxSum_;
16    }
17
18 private:
19     int CountPathSum(TreeNode* root)
20     {
21         if(root->left == 0 && root->right == 0)
22         {
23             if(firstValue_ || root->val > maxSum_)
24             {
25                 maxSum_ = root->val;
26                 firstValue_ = false;
27             }
28             return root->val;

```

```

29     }
30     else
31     {
32         int left = 0;
33         int right = 0;
34         if(root->left)
35         {
36             left = CountPathSum(root->left);
37         }
38
39         if(root->right)
40         {
41             right = CountPathSum(root->right);
42         }
43
44         int currentBest = 0;
45         int sumInPah = 0;
46
47         if(left > 0 && right > 0)
48         {
49             currentBest = left + right;
50
51             sumInPah = left > right ? left : right;
52         }
53         else if(left > 0)
54         {
55             currentBest = left;
56             sumInPah = left;
57         }
58         else if(right > 0)
59         {
60             currentBest = right;
61             sumInPah = right;
62         }
63         else
64         {
65             if(!root->left)
66             {
67                 currentBest = right;
68             }
69             else if(!root->right)
70             {
71                 currentBest = left;
72             }

```

```

73         else
74         {
75             currentBest = left > right ? left : right;
76
77         }
78         sumInPah = currentBest;
79     }
80
81         //前面已做只取正值的处理,如果还小于零说明
两个都是负数
82         if(sumInPah < 0)
83         {
84             sumInPah = root->val;
85         }
86         else
87         {
88             sumInPah += root->val;
89         }
90
91         if(currentBest < 0)
92         {
93             currentBest = root->val;
94         }
95         else
96         {
97             currentBest += root->val;
98         }
99
100         if(currentBest > maxSum_)
101         {
102             maxSum_ = currentBest;
103         }
104
105         return sumInPah;
106     }
107 }
108
109 int maxSum_;
110 bool firstValue_;
111 };

```

提交后 Judge Small 和 Judge Large 都顺利通过。

