注意溢出，正负

# Java

## 同步synchronized

1、synchronized关键字的作用域有二种：

1）同步非静态方法，锁对象实例。

synchronized aMethod(){}防止多个线程同时访问这个对象的synchronized方法。

如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法。

不同的对象实例的synchronized方法是不相干扰的，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；

2）同步静态方法，锁类本身。

其他线程不能同时访问相同类的不同实例对象的静态同步方法。

2、除了方法前用synchronized关键字，synchronized关键字还可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。用法是: synchronized(this){/\*区块\*/}，它的作用域是当前对象；

3、synchronized关键字是不能继承的，也就是说，基类的方法synchronized f(){} 在继承类中并不自动是synchronized f(){}，而是变成了f(){}。继承类需要你显式的指定它的某个方法为synchronized方法。

pulbic class Something(){

public synchronized void isSyncA(){}

public synchronized void isSyncB(){}

public static synchronized void cSyncA(){}

public static synchronized void cSyncB(){}

}

       那么，加入有Something类的两个实例a与b，那么下列组方法何以被1个以上线程同时访问呢

a. x.isSyncA()与x.isSyncB()

b. x.isSyncA()与y.isSyncA()

c. x.cSyncA()与y.cSyncB()

d. x.isSyncA()与Something.cSyncA()

a，都是对同一个实例的synchronized域访问，因此不能被同时访问

b，是针对不同实例的，因此可以同时被访问

c，因为是static synchronized，所以不同实例之间仍然会被限制,相当于Something.isSyncA()与 Something.isSyncB()了，因此不能被同时访问。

d，可以被同时访问的，synchronzied实例方法与synchronzied类方法锁定（lock）不同。

## static语句

类加载时只执行一次。

调用类的静态常量的时候，是不会加载类的，即不会执行static{}语句块。

当一个类中有多个static{}的时候，按照static{}的定义顺序，从前往后执行。

## 单例

public class Singleton{

private **static** Singleton \_instance = null;

**private** Singleton(){}

public **static** Singleton getInstance(){

if( null == \_instance){

**synchronized** (Singleton.class){

if( null == \_instance){

\_instance = new Singleton();

}

}

}

return \_instance;

}

}

不同ClassLoader创建多个单例，避免不同ClassLoader加载同一单例

## 枚举

enum Signal {

    GREEN, YELLOW, RED

}

public class TrafficLight {

    Signal color = Signal.RED;

    public void change() {

        switch (color) {

        case RED:

            color = Signal.GREEN;

            break;

        case YELLOW:

            color = Signal.RED;

            break;

        case GREEN:

            color = Signal.YELLOW;

            break;

        }

    }

}

## 抽象类vs接口

抽象类可以有数据成员和非抽象方法，接口只能有final static数据成员，所有方法必须抽象。

抽象类是单一继承关系，接口多重继承。接口中可以没有任何方法，如Serializable。

抽象类赋予默认行为。

## HashMap vs HashTable

1、父类不同

HashTable继承自Dictionary类，HashMap继承自Map接口。

2、线程安全

HashTable同步，HashMap不同步

3、Key和Value是否允许有null值

HashTable不允许，HashMap允许。

4、方法不同

HashTable包含contains方法。

## 生产者消费者

//一个数

public class Product{

public int value = -1;

}

public class Producer extends Thread{

private Product product;

private Random rand = new Random();

public Producer(Product product){

this.product = product;

}

public void run(){

while(true){

try{

synchronized(product){

while(product.value>0){product.wait();}

product.value = rand.nextInt(101);

product.notifyAll();

}

}catch(Exception e){

e.printStackTrace();

}

}

}

}

public class Consumer extends Thread{

private Product product;

public Consumer(Product product){

this.product = product;

}

public void run(){

while(true){

try{

synchronized(product){

while(product.value < 0){product.wait();}

System.out.println(product.value);

product.notifyAll();

product.value = -1;

}

}catch(Exception e){

e.printStackTrace();

}

}

}

}

public class ProducerConsumer{

public static void main(String[] args){

Product product = new Product();

Producer producer = new Producer(product);

Consumer consumer = new Consumer(product);

producer.start();

consumer.start();

}

}

//多个数

public class Product{

private String[] buffer;

private int tail;

private int head;

private int count;

public Product(int capacity){

this.buffer = new String[capacity];

this.head = 0;

this.tail = 0;

thihs.count = 0;

}

public synchronized void produce(String str){

while(count >= buffer.length){

wait();

}

buffer[tail]=str;

tail = (tail+1)%buffer.length;

count++;

notifyAll();

}

public synchronized String consume(){

while(count<=0){

wait();

}

String str = buffer[head];

head = (head+1)%buffer.length;

count--;

notifyAll();

return str;

}

}

public class Producer extends Thread{

private Random rand = new Random();

private Product product;

public Producer(Product product){

this.product = product;

}

public void run(){

while(true){

product.produce(rand.nextInt()+””);

}

}

}

public class Consumer extends Thread{

private Product product;

public Consumer(Product product){

this.product = product;

}

public void run(){

while(true){

System.out.println(product.consume());

}

}

}

## wait/sleep/notify

wait()是Object类中定义的方法，与notify/notifyAll()在一起成对使用，要释放对象锁。

sleep()是Thread类中的一个静态方法，不会释放对象锁

notify()唤醒正在等待的单个线程

notifyAll()唤醒所有正在等待的线程

wait、notify、notifyAll必须在synchronized修饰的代码块中执行，否则会在运行的时候抛出IllegalMonitorStateException异常

## StringBuffer&StringBuilder

String 字符串常量

StringBuffer 字符串变量（线程安全）

StringBuilder 字符串变量（非线程安全）

大部分情况下，StringBuilder>StringBuffer>String

## Math.floor

Math.floor把数字变小，Math.ceil把数字变大

Math.floor(99.1) = 99.0

Math.floor(-99.1) = -100.0

Math.ceil(99.1) = 100.0

Math.ceil(-99.1) = -99.0

## JSP内置对象

输出输入对象:request对象、response对象、out对象

通信控制对象:pageContext对象、session对象、application对象

Servlet对象:page对象、config对象

错误处理对象:exception对象

## 面向对象的3个基础和5个基本原则

面向对象的3个基本要素：封装、继承、多态

面向对象的5个基本设计原则：

单一职责原则（Single-Resposibility Principle）

其核心思想为：一个类，最好只做一件事，只有一个引起它的变化

开放封闭原则（Open-Closed principle）

其核心思想是：软件实体应该是可扩展的，而不可修改的。也就是，对扩展开放，对修改封闭的。

Liskov替换原则（Liskov-Substituion Principle）

其核心思想是：子类必须能够替换其基类。

依赖倒置原则（Dependecy-Inversion Principle）

其核心思想是：依赖于抽象。

接口隔离原则（Interface-Segregation Principle）

其核心思想是：使用多个小的专门的接口，而不要使用一个大的总接口。

# C/C++

## itoa实现

void itoa(int value, char \*str){

int start = 0;

if(value<0){

str[0]=’-’;

value = 0-value;

start = 1;

}

int i=start;

while(value){

str[i++] = value%10 +’0’;

value /= 10;

}

int j=start;

while(j<i){

char tmp = str[i];

str[i--] = str[j];

str[j++] = tmp;

}

}

## malloc/free vs new/delete

相同点：都可用于申请动态内存和释放内存

不同点：

malloc/free是C/C++语言的标准库函数，new/delete是C++的运算符。

malloc/free无法自动执行构造函数和析构函数。

调用方式不同。

# 操作系统

进程:资源分配的最小单位

线程:CPU调度和执行的最小单位.

1G->1024M->1024\*1024KB->1024\*1024\*1024B->1024\*1024\*1024\*8bit

# 数据库

## 范式

1NF：不可分割

2NF：不存在部分依赖

3NF：不存在传递依赖，即X->Y，Y->Z

BCNF：

## 数据库事务

原子性，持久性，独立性，一致性

## SQL四种语言

DML：表内容操作，SELECT，INSERT，UPDATE，DELETE，MERGE，CALL，EXPLAIN PLAN，LOCK TABLE

DDL：表结构操作CREATE，ALTER，DROP，TRUNCATE，COMMENT，RENAME

DCL：表权限操作，GRANT ，REVOKE

TCL（Transaction Control Language）事务控制语言：SAVEPOINT 设置保存点，ROLLBACK  回滚，SET TRANSACTION

group by在where之后，order by在所有之后

Insert into Table2(field1,field2,...) select value1,value2,... from Table1

SELECT vale1, value2 into Table2 from Table1

## 索引

索引是对数据库表中一列或多列的值进行排序的一种结构。

索引分为聚簇索引和非聚簇索引两种，聚簇索引 是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

经常需要排序的列，经常使用在WHERE子句中的列上建索引

优点：加快对表中记录的查找或排序

缺点：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间

## left join&right join

left join：以左表为准，左表全部显示，右表搜索符合条件的记录，没有则为NULL。

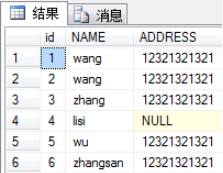
right join相反。

inner join：只显示两边都符合条件的记录。

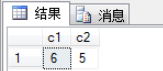
full join：两边都显示，找不到则以NULL代替。

## count

count(\*)和count(字段名)区别



SELECT COUNT(\*) c1 ,COUNT(ADDRESS) c2 FROM test



count(\*)统计的是结果集的总条数，count(字段名)统计的是该字段值不为null的总条数

# 网络

## 五层

应用层：HTTP，DNS，ICMP（ping），SMTP

传输层：TCP，UDP

网络层：IP

数据链路层

物理层

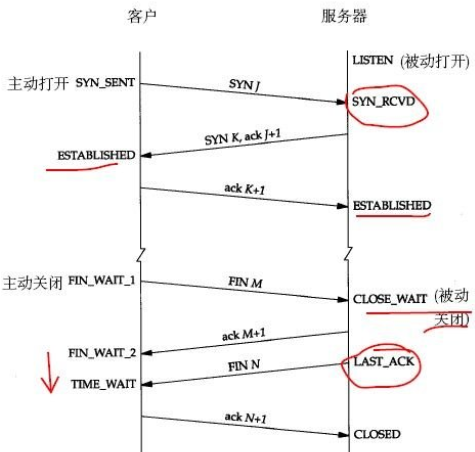
ARP：IP->MAC，DNS：域名->IP

## TCPvsUDP

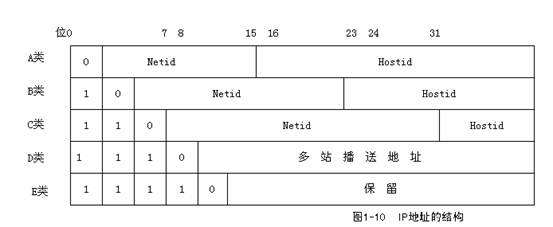
TCP：传输控制协议，面向连接、可靠的字节流服务。建立TCP连接，提供超时重发、丢弃重复数据等功能。

UDP：用户数据报协议。面向数据报的传输层协议。不可靠，不建立连接，传输速度快。

## TCP的三次握手



## IP地址



# 算法

## 插入排序

平均最差O(n2)，最好O(n)

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

int i,j,temp;

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

**for(int j=i;( j>0) && (a[j]<a[j-1]); j--){**

temp = a[j];

a[j] = a[j-1];

a[j-1] = temp;

}

}

}

## 选择排序

不稳定，O(n2)

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

int i, j , temp;

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

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

if(a[i]>a[j]){

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

## 冒泡排序

平均最差O(n2)，最好O(n)

void bubbleSort(int a[],int n)

{

    int i,j,temp;

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

**{**

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

        {

            if(a[j+1]<a[j])

            {

              temp=a[j+1];

              a[j+1]=a[j];

              a[j]=temp;

             }

        }

    }

}

## 快速排序

时间复杂度：最好平均O(nlog2n)，最坏O(n2)

空间复杂度：O(log2n)

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

int key = a[low], i=low, j=high;

while(i<j){

while(a[j]>key&&i<j) j--;

if(i<j) a[i++] = a[j];

while(a[i]<key&&i<j) i++;

if(i<j) a[j--] = a[i];

}

a[i]=key;

quickSort(a, low, i-1);

quicksort(a, i+1; high);

}

## 二分查找

int binSearch(int a[], int n, int key){

int low = 0, high = n-1, mid;

while(low<=high){

mid = (low+high)/2;

if(a[mid]<key){

low = mid+1;

}else if(a[mid]>key){

high = mid-1;

}else{

return mid;

}

}

return -1;

}

## 二分排序

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

int i,j,low,high,mid,temp;

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

temp = a[i];

low = 0;

high = i-1;

while(low<=high){

mid = (low+high)/2;

if(a[mid]>temp){

high = mid-1;

}else{

low = mid+1;

}

}

for(int j=i-1; j>high; j--)

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

}

}

## 求质数

boolean suShu(int n){

int m = sqrt(n);

for(int i=2; i<m; i++){

if(n%i == 0) return false;

}

return true;

}

## 斐波那契数列

int f(int n){

if(n == 1 || n == 2){

return 1;

}

return f(n-1)+f(n-2);

}

## 找数组中第二大的数

int findSecondMax(int a[], int n){

int max[] = new int[2];

if(n<2) return -1;

if(a[0]>a[1]){

max[0] = a[0];

max[1] = a[1];

}else{

max[0] = a[1];

max[1] = a[0];

}

for(int i=2; i<n; i++){

if(a[i]>max[0]){

max[0] = a[i];

max[1] = max[0];

}else if(a[i] > max[1]){

max[1]=a[i];

}

}

return max[1];

}

## 杨氏矩阵问题

如果一个矩阵每一行每一列都严格单调递增，我们称该矩阵为杨氏矩阵（Young Tableau）。对于杨氏矩阵(a[m][n])，通常会涉及两个问题：(1) 怎样在杨氏矩阵中查找某个元素X？(2) 怎样在杨氏矩阵找第k大的数？

(1) 问题1求解

【方案一】

< 二分查找>

对于杨氏矩阵，由于每行每列均是有序的，则可以于矩阵采用二分查找。具体方法是：

对于当前子矩阵a[i][j]~a[s][t]，中间元素为a[(i+s)/2][(j+t)/2]，如果a[(i+s)/2][(j+t) /2]==x，则找出该元素；如果a[(i+s)/2][(j+t)/2] > x，则在子矩阵a[i][j]~a[(i+s)/2][(j+t)/2]中查找；如果a[(i+s)/2][(j+t)/2] < x，则在三个子矩阵：a[i][(j+t)/2]~ a[(i+s)/2][t]，a[(i+s)/2][(j+t)/2]~ a[s][t]和a[(i+s)/2][j]~ a[s][ (j+t)/2]中查找。该算法的递归式为f(mn)=3f(mn/4)+O(1)，根据主定理知，时间复杂度为：O((mn)^(log4(3)))。

【方案二】

<类堆查找法>

杨氏矩阵具有明显的堆特征：从矩阵的右上角出发（从左下角出发思路类似），对于元素a[i][j]，如果a[i][j]==x，则找到元素x，直接返回； 如果a[i][j] > x，则向下移动，即继续比较a[i+1][j]与x；如果a[i][j] < x，则向左移动，即继续比较a[i][j-1]与x。该算法的时间复杂度是O(m+n)，代码如下：

bool find\_in\_YoungTableau(int\*\* a, int m, int n, int x) {

assert(a != NULL && m > 0 && n > 0);

int row, col;

row = 0;

col = n-1;

while(row <= m-1 && col >= 0) {

if(a[row][col] == x)

return true;

else if(a[row][col] > x)

col--;

else

row++;

}

return false;

}

(2) 问题2求解   
【方案一】

<小顶堆法>

首先将a[0][0]加入小顶堆，然后每次从小顶堆中取出最小元素，并加入比该元素大且与之相邻的两个元素（对于a[0][0]，则需要加入a[0] [1]和a[1][0]），直到取出第k个元素，需要注意的是，需要采用额外空间记录某个元素是否已经加入到小顶堆以防止重复加入。

【方案二】

<二分枚举+类堆查找>   
首先，二分枚举找到一个数x，它比杨氏矩阵中k个数大；然后，利用类堆查找法找到刚好小于x的元素。该算法的时间复杂度为O((m+n)log(mn))，但不需要额外存储空间。代码如下：

int find\_kth\_in\_YoungTableau(int\*\* a, int m, int n, int k) {

int low = a[0][0];

int high = a[m-1][n-1];

int order = 0;

int mid = 0;

do {

mid = (low + high) >> 1;

order = get\_order(a, m, n, mid);

if(order == k)

break;

else if(order > k)

high = mid - 1;

else

low = mid + 1;

} while(1); //二分枚举整，找出正好大于k的一个整数 mid

int row = 0;

int col = n - 1;

int ret = mid;//这个地方有些问题

while(row <= m-1 && col >= 0) { //找出比mid小的最大数

if(a[row][col] < mid) {//并且应该考虑到矩阵中可能会存在巧好等于mid的数，然后就直接返回就好了

ret = (a[row][col] > ret) ? a[row][col] : ret;

row++;

} else {

col--;

}

}

return ret;

}

//整数k在矩阵中的排名

int get\_order(int\*\* a, int m, int n, int k) {

int row, col, order;

row = 0;

col = n-1;

order = 0;

while(row <= m-1 && col >= 0) {

if(a[row][col] < k) {

order += col + 1;

row++;

} else {

col--;

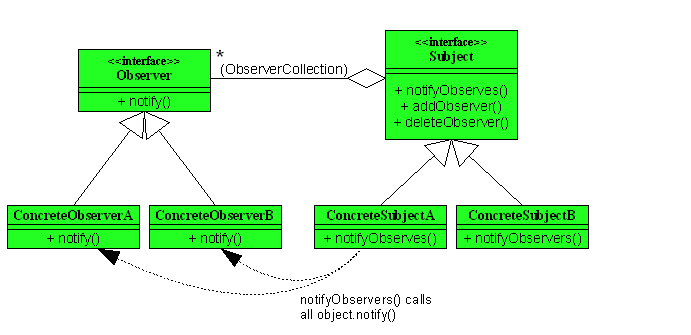
}

}

return order;

}

## 监听者模型



public interface Subject {

public void addListener(Listener listener);

public void deleteListener(Listener listener);

public void notifyListener();

}

public class Lights implements Subject{

private ArrayList lights;

private String whichLight;

private String currentLight = "green";

private int time;

public Lights(){

lights = new ArrayList();

}

@Override

public void addListener(Listener listener){

lights.add(listener);

}

@Override

public void deleteListener(Listener listener){

int index = lights.indexOf(listener);

if( index != -1){

lights.remove(index);

}

}

@Override

public void notifyListener(){

int size = lights.size();

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

Listener listener = (Listener)lights.get(i);

listener.updateSignal(whichLight,time);

}

}

public void setLight(String whichlight,int time){

this.whichLight = whichlight;

this.time = time;

//检查状态是否发生了变化

check();

this.currentLight = whichLight;

}

private void check(){

if(! this.currentLight.equals(this.whichLight)){

notifyListener();   
}

}

}

public interface Listener {

public void updateSignal(String whichLight,int time);

}

public class RedLightListener implements Listener{

@Override

public void updateSignal(String whichLight, int time){

if(whichLight.equals("red")){

System.out.println("红灯亮了，禁止通行");

System.out.println("持续时间： "+time);

}

}

}

public class YellowLightListener implements Listener{

@Override

public void updateSignal(String whichLight,int time){

if(whichLight.equals("yellow")){

System.out.println("黄灯亮了，请稍等");

System.out.println("持续时间： "+ time);

}

}

}

public class GreenLightListener implements Listener{

@Override

public void updateSignal(String whichLight,int time){

if(whichLight.equals("green")){

System.out.println("绿灯亮了，请通过");

System.out.println("持续时间： "+time);

}

}

}

public class ObserverPattern {

public static void main(String[] args) {

Lights lights = new Lights();

YellowLightListener yellow = new YellowLightListener();

GreenLightListener green = new GreenLightListener();

RedLightListener red = new RedLightListener();

lights.addListener(green);

lights.addListener(red);

lights.addListener(yellow);

lights.setLight("red",30);

lights.setLight("yellow", 10);

lights.setLight("green", 20);

}

}

## 寻找兄弟单词

一个单词单词字母交换，可得另一个单词，如army->mary，成为兄弟单词。提供一个单词，在字典中找到它的兄弟。描述数据结构和查询过程。

使用hash\_map和链表

（1）首先定义一个key，使得兄弟单词有相同的key，不是兄弟的单词有不同的key。例如，将单词按字母从小到大重新排序后作为其key，比如bad的key为abd，good的key为dgoo。

（2）使用链表将所有兄弟单词串在一起，hash\_map的key为单词的key，value为链表的起始地址。

（3）开始时，先遍历字典，将每个单词都按照key加入到对应的链表当中。

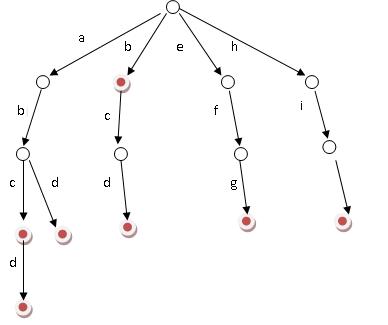
（4）当需要找兄弟单词时，只需求取这个单词的key，然后到hash\_map中找到对应的链表即可。  这样创建hash\_map时时间复杂度为O(n)，查找兄弟单词时时间复杂度是O(1)。

## 字典树

3个基本性质：

* 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
* 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
* 每个节点的所有子节点包含的字符都不相同。

假设有b，abc，abd，bcd，abcd，efg，hii 6个单词，字典树如下图：



#define MAX 26

typedef struct Trie{

Trie \*next[MAX];

int v;

};

Trie \*root;

next是表示每层有多少种类的数，如果只是小写字母，则26即可，若改为大小写字母，则是52，若再加上数字，则是62了，这里根据题意来确定。

void createTrie(char\* str){

int len = strlen(str);

Trie \*p = root, \*q;

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

int id = str[i]-‘0’;

if(p->next[id] == NULL){

q = (Trie\*)malloc(sizeof(Trie));

q->v = 1;

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

q->next[j] = NULL;

}

p->next[id] = q;

p = p->next[id];

}

}else{

p->next[id]->v++;

p = p->next[id];

}

p->v = -1;

}

int findTrie(char \*str){

int len = strlen(str);

Trie \*p = root;

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

int id = str[i]-‘0’;

p = p->next[id];

if(p == NULL){ //若为空集，表示不存以此为前缀的串

return 0;

}else if(p->v == -1){ //字符集中已有串是此串的前缀

return -1;

}

}

return 1; //此串是字符集中某串的前缀

}

int dealTrie(Trie\* T)

{

int i;

if(T==NULL)

return 0;

for(i=0;i<MAX;i++)

{

if(T->next[i]!=NULL)

deal(T->next[i]);

}

free(T);

return 0;

}

## 树

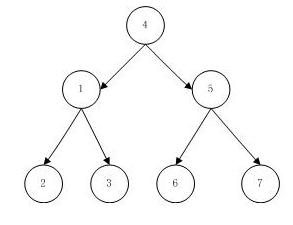
typedef struct TNode{

int val;

struct TNode \*pLeft;

struct TNode \*pRight;

}

[](http://static.oschina.net/uploads/space/2012/0516/190747_ksto_100374.jpg)

### 先序遍历

递归算法：

void preTraverse(TNode \*pRoot){

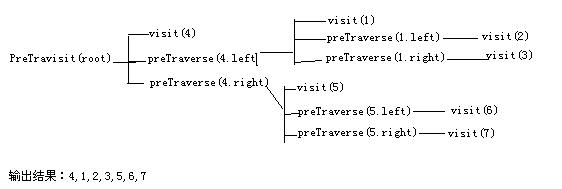
if(!pRoot) return;

visit(pRoot);

preTraverse(pRoot->pLeft);

preTraverse(pRoot->pRight);

}



非递归算法：

void preNoRecTraverse(TNode \*pRoot){

stack<TNode\*> s;

TNode \*p = pRoot;

s.push(p);

while(p || !s.empty()){

p = s.top();

s.pop();

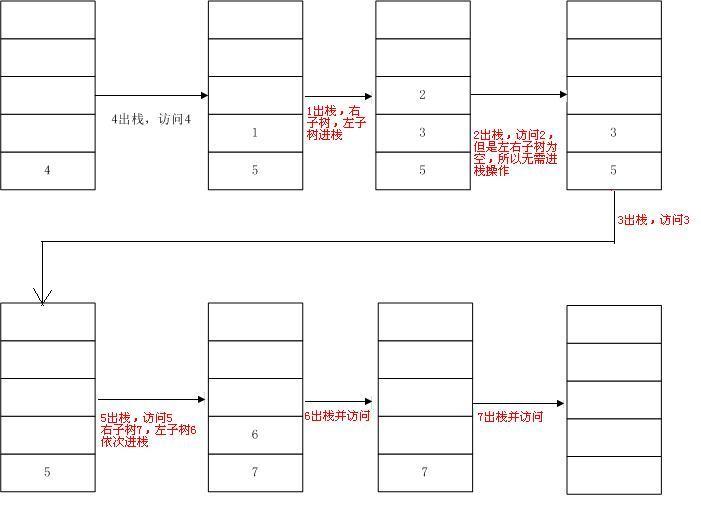
visit(p);

if(p->pRight) s.push(p->pRight);

if(p->pLeft) s.push(p->pLeft);

}

}



### 中序遍历

递归算法：

void midTraverse(TNode \*pRoot){

if( !pRoot ) return;

midTraverse(pRoot->pLeft);

visit(pRoot);

midTraverse(pRoot->pRight);

}

非递归算法：

void midNoRecTraverse(TNode \*pRoot){

stack<TNode\*> s;

TNode \*p = pRoot;

s.push(p);

while(p || !s.empty()){

while(!p){

s.push(p);

p = p->pLeft;

}

p = s.top();

s.pop();

visit(p);

p = p->pRight;

}

}

### 后序遍历

递归算法：

void postTraverse(TNode \*pRoot){

if(!pRoot) return ;

postTraverse(pRoot->pLeft);

postTraverse(pRoot->pRight);

visit(pRoot);

}

非递归算法：