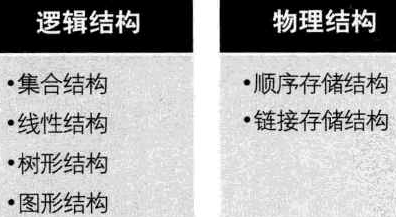
窗体底端

数据结构以某种形式将数据组织在一起的集合，不仅存储数据，还支持访问和处理数据的操作。

算法是为求解一个问题需要遵循的、被清楚指定的简单指令的集合。

正确性、可读性、健壮性、高效率 、低存储量



**一、线性表**

n个数据元素的有限序列。

实现方式:

链表存储线性表的元素，即用一组任意的存储单元存储线性表的数据元素（存储单元可以是连续的，也可以是不连续的）。

* 1. **数组实现**

用一组连续的存储单元依次存储线性表的数据元素.

数组一旦创建，大小就无法改变，但当数组容量不够，可创建新的大数组来替换当前数组。

* **1.1.1 数组扩容**
* **创建一个更大的数组来替换当前数组**

|  |
| --- |
| int[] oldArray = new int[10];  int[] newArray = new int[20];  for (int i = 0; i < oldArray.length; i++) {  newArray[i] = oldArray[i];  }  // 也可以使用System.arraycopy方法来实现数组间的复制  // System.arraycopy(oldArray, 0, newArray, 0, oldArray.length);  oldArray = newArray; |

* **1.1.2 插入**

**在数组位置index上添加(插入)元素e**

**思路:**

* 如插入位置不合理，抛出异常
* 如线性表长度>数组长度，抛出异常或动态扩容
* 从最后一个元素开始向前遍历到index位置，分别将它们后移一个位置。

|  |
| --- |
| //oldArray 表示当前存储元素的数组  //size 表示当前元素个数  public void add(int index, int e) {  if (index > size || index < 0) {  System.out.println("位置不合法...");  }  //如果数组已经满了 就扩容  if (size >= oldArray.length) {  // 扩容函数可参考代码1  }  for (int i = size - 1; i >= index; i--) {  oldArray[i + 1] = oldArray[i];  }  //将数组elementData从位置index的所有元素往后移一位  // System.arraycopy(oldArray, index, oldArray, index + 1,size - index);  oldArray[index] = e;  size++;  } |

* **1.1.3 删除**

思路：

|  |
| --- |
| 如删除位置不合理，抛出异常  取出删除元素  从删除元素位置开始遍历到最后一个元素，分别将它们向前移一个位置。  表长减一 |

* **1.1.4 优缺点**

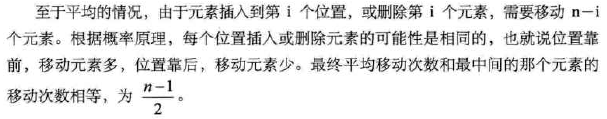
具体可参考ArrayList集合类源码。

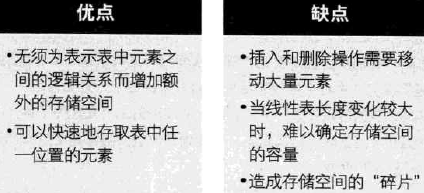
* 数组实现的线性表优点：可通过下标访问或修改元素，较高效

|  |
| --- |
| 存、读数据，O(1) |

* 缺点：插入和删除花费开销较大。

|  |
| --- |
| 最好:如元素要插入到最后一个位置或删除最后一个元素，O(1)  最坏: 如元素要插入到第一个位置或删除第一个元素，O(n)  平均：O(n) |

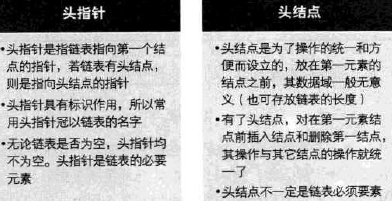




* 1. **链表**

物理存储单元上非连续、非顺序，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

链表由一系列节点组成，节点不必在内存中相连。



节点：数据部分Data + 链部分Next。Next指向下一个节点。当添加或删除时，只需改变相关节点的Next的指向，效率很高。



单链表的结构

下面主要用代码来展示链表的一些基本操作，需要注意的是，这里主要是以单链表为例，暂时不考虑双链表和循环链表。

* + 1. **代码3 链表的节点**

|  |
| --- |
| class Node<E> {  E item;  Node<E> next;  //构造函数  Node(E element) {  this.item = element;  this.next = null;  }  } |

* + 1. 代码4 头/尾节点初始化

定义好节点后，使用前对头节点和尾节点初始化

|  |
| --- |
| //头节点和尾节点都为空 链表为空  Node<E> head = null;  Node<E> tail = null; |

* + 1. 代码5 空链表创建一个新节点

//创建一个新节点 并让head指向此节点

head = new Node("nodedata1");

//让尾节点也指向此节点

tail = head;

* + 1. 代码6 链表追加一个节点

//创建新节点 同时和最后一个节点连接起来

tail.next = new Node("node1data2");

//尾节点指向新的节点

tail = tail.next;



* + 1. 代码7 顺序遍历链表

Node<String> current = head;

while (current != null) {

System.out.println(current.item);

current = current.next;

}

* + 1. 代码8 倒序遍历链表

|  |
| --- |
| static void printListRev(Node<String> head) {  //倒序遍历链表主要用了递归的思想(类似栈的思想)  //执行顺序: printListRev(head.next); ~ if (head != null) ~ printListRev(head.next); ~ //if (head != null) (此时不符合了，即达到边界条件，结束。开始倒序取之前放入//“栈”的函数段执行)  if (head != null)  printListRev(head.next);  System.out.println(head.item);  }  } |

* + 1. 代码 单链表反转

|  |
| --- |
| //单链表反转 主要是逐一改变两个节点间的链接关系来完成  static Node<String> revList(Node<String> head) {  if (head == null) {  return null;  }  Node<String> nodeResult = null;  Node<String> nodePre = null;  Node<String> current = head;  while (current != null) {  Node<String> nodeNext = current.next;  if (nodeNext == null) {  nodeResult = current;  }  current.next = nodePre;  nodePre = current;  current = nodeNext;  }  return nodeResult;  } |

还有像获取指定元素，移除元素等操作

* + 1. 单链表读取

最坏O(N)

链表的实现还有其它的方式:

* **循环单链表** 链表的最后一个节点指向第一个节点，整体构成一个链环。
* **双向链表** 节点中包含两个指针部分，一个指向前驱元，一个指向后继元，JDK中LinkedList集合类的实现就是双向链表。
* **循环双向链表** 最后一个节点指向第一个节点。

**二、栈与队列**

较特殊的线性表：

* 栈，访问、插入和删除元素只能在栈顶进行
* 队列，元素只能从队尾插入，从队头访问和删除。

**2.1 栈**

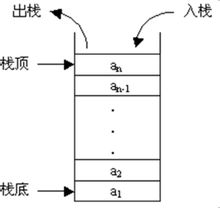
2.1.1 概述

限制插入和删除只能在一个位置(表的末端，栈顶)上进行的表。

有时又叫LIFO(Last In First Out)表，即后进先出。

push(进栈): 插入

pop(出栈): 删除最后一个元素。



2.1.2 栈的模型



答案C

因为栈也是一个表，所以任何实现表的方法都能实现栈。

|  |
| --- |
| Public class Stack<E> extends Vector<E> { |

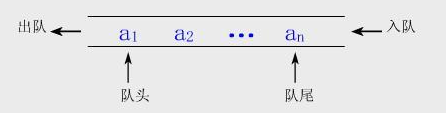
可使用LinkedList来进行栈的所有操作。

* 1. **队列**

2.2.1 概述

特殊的线性表，只许在表前端（队头 front）删除，在表后端（队尾 rear）插入

和栈一样，队列是一种操作受限制的线性表。



可用链表来实现队列

2.2.2 代码9 简单实现队列类（利用LinkedList）

|  |
| --- |
| public class MyQueue<E> {  private LinkedList<E> list = new LinkedList<>();  // 入队  public void enqueue(E e) {  list.addLast(e);  }  // 出队  public E dequeue() {  return list.removeFirst();  }  } |

* 普通队列: 先进先出
* 优先队列: 元素被赋予优先级。访问元素时，具有最高优先级的元素最先被删除。

Java集合框架中，类PriorityQueue是优先队列的实现类

**三、树与二叉树**

重要的非线性数据结构，以树和二叉树最为常用。

**3.1 树**

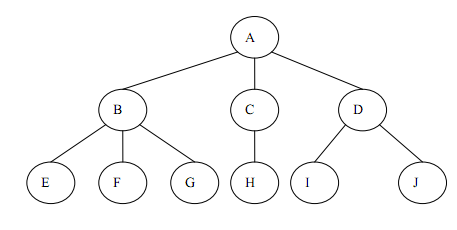
**树** 是由n（n>=1）个有限节点组成一个具有层次关系的集合。

特点：每个节点有零个或多个子节点；

**根**节点: 没有父节点的节点

每一个非根节点有且只有一个**父节点** ；

除了根节点外，每个子节点可以分为多个不相交的子树。



**3.2 二叉树**

**3.2.1 定义**

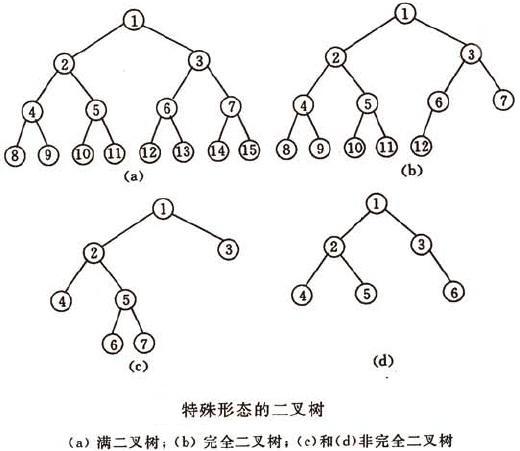
二叉树:每个节点最多有两棵子树的树结构。

子树被称作“左子树”和“右子树”。

常被用于实现二叉查找树和二叉堆。

**3.2.2 相关性质**

* 每个结点至多2棵子树
* 子树有左右之分，次序不能颠倒
* 第i层至多有2^(i-1)个结点；(2^0 2^1 2^2…)
* 深度为k的二叉树至多有2^k-1个结点.(2^0 + 2^1 + 2^2 + …+2^(k-1)= 2^k -1)
* 深度为k，且有2^k-1个节点的二叉树称为 **满二叉树** ；
* 深度为k，有n个节点的二叉树，当且仅当其每一个节点都与深度为k的满二叉树中，序号为1至n的节点对应时，称之为 **完全二叉树** 。



**3.2.3 三种遍历方法**

在树中查找具有某种特征的节点，或对树中全部节点进行某种处理，就涉及到二叉树的遍历。

二叉树主要由3个基本单元组成，根节点、左子树和右子树。

如限定先左后右，根据这三个部分遍历顺序不同，可分为先序、中序和后续遍历。

* + - 1. **先序遍历** (根-左-右)

若二叉树为空，则空操作

否则先访问根节点，再先序遍历左子树，最后先序遍历右子树。

* + - 1. **中序遍历 （左-根-右）**

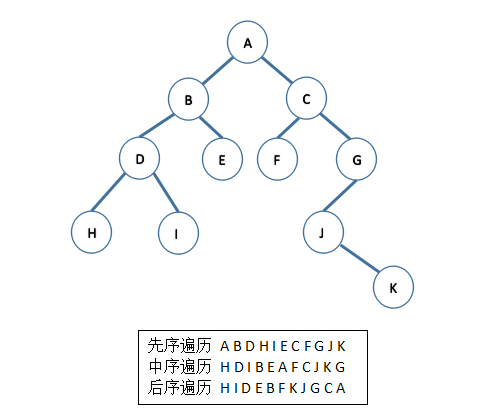
若二叉树为空，则空操作

否则先中序遍历左子树，再访问根节点，最后中序遍历右子树。

* + - 1. **后序遍历 （左右根）**

若二叉树为空，则空操作

否则先后序遍历左子树访问根节点，再后序遍历右子树，最后访问根节点。



**3.3 树和二叉树的区别**

(1) 二叉树每个节点最多有2个子节点，树则无限制。

(2) 二叉树中节点的子树分为左子树和右子树，即使某节点只有一棵子树，也要指明该子树是左子树还是右子树，即二叉树是有序的。

(3) 树决不能为空，至少有一个节点；而二叉树可为空。

**3.4 二叉查找树**

**3.4.1 定义**

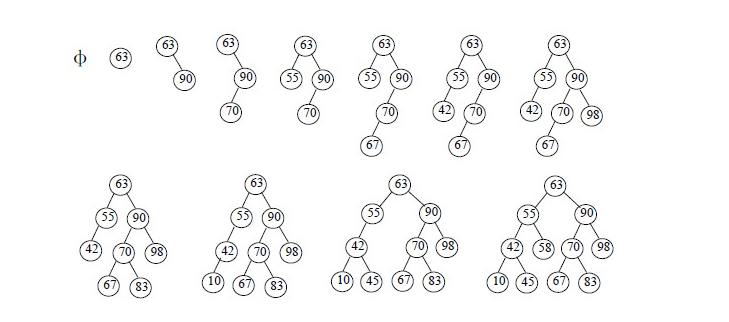
* 就是二叉排序树，也叫二叉搜索树。
* 二叉查找树或为空树，或是具有下列性质的二叉树：

(1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；（左<根）

(2) 若右子树不空，则右子树上所有结点的值均大于它的根结点的值；（根<右）

(3) 左、右子树也分别为二叉排序树；

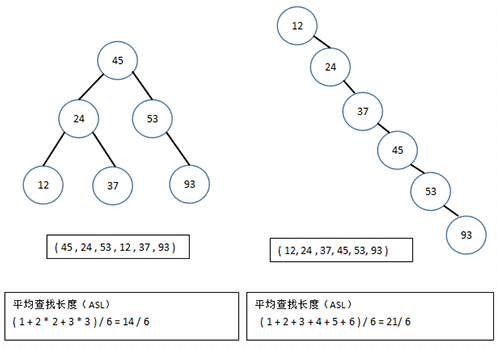
(4) 没有键值相等的结点。



典型的二叉查找树的构建过程

**3.4.2性能分析**

对二叉查找树来说，当给定值相同但顺序不同，所构建的二叉查找树形态不同



不同形态, 平衡二叉树的ASL不同

含有n个节点的二叉查找树的平均查找长度和树的形态有关。

最坏情况下，当先后插入的关键字有序时，构成的二叉查找树蜕变为单支树，树的深度为n，其平均查找长度(n+1)/2(和顺序查找相同），

最好的情况是二叉查找树的形态和折半查找的判定树相同，其平均查找长度和log2(n)成正比。

平均情况下，二叉查找树的平均查找长度和logn是等数量级的，所以为了获得更好的性能，通常在二叉查找树的构建过程需要进行“平衡化处理”，之后我们将介绍平衡二叉树和红黑树，这些均可以使查找树的高度为O(log(n))。

**3.4.3二叉树的节点先序/中序/后序遍历 代码**

二叉查找树的三种遍历都可以直接用递归的方法来实现：

* TreeNode.java

|  |
| --- |
| package tree;  public class TreeNode<E> {    E element;  public TreeNode<E> left;  public TreeNode<E> right;    public TreeNode(E element) {  this.element = element;  }    //先序遍历  public void preorder(TreeNode<E> root) {  if(root == null) {  return;  }    System.out.println(root.element + "");    preorder(root.left);  preorder(root.right);    }    //中序遍历  public void inorder(TreeNode<E> root) {  if(root == null) {  return;  }    inorder(root.left);    System.out.println(root.element + "");    inorder(root.right);  }    //代码14 后序遍历  public void postorder(TreeNode<E> root) {  if (root == null) {  return;  }  postorder(root.left);  postorder(root.right);  System.out.println(root.element + " ");  }  } |

测试代码：

|  |
| --- |
| package test;  import org.junit.Test;  import tree.TreeNode;  public class TestTreeNode {  @Test  public void test() {  TreeNode treeNode = new TreeNode("rootdata1");  treeNode.left = new TreeNode("leftdata1");  treeNode.right = new TreeNode("rightdata1");    treeNode.preorder(treeNode);  //treeNode.inorder(treeNode);  //treeNode.postorder(treeNode);  }  } |

**3.4.4 二叉查找树的简单实现**

|  |
| --- |
| package tree.binsearchtree;  public class MyBinSearchTree<E extends Comparable<E>> {    //根  private TreeNode<E> root;    // 默认构造函数  public MyBinSearchTree(TreeNode treeNode) {  this.root = treeNode;  }    // 二叉查找树的搜索  public boolean search(E element) {    TreeNode<E> current = root;  while (current != null) {  if(element.compareTo(current.element) < 0) {  current = current.left;  } else if (element.compareTo(current.element) > 0) {  current = current.right;  } else {  return true;  }  }  return false;  }    // 二叉查找树的插入  public boolean insert(E element) {  // 如果之前是空二叉树 插入的元素就作为根节点  if(root == null) {  root = createNewNode(element);  } else {  // 否则就从根节点开始遍历 直到找到合适的父节点  TreeNode<E> parent = null;  TreeNode<E> current = root;    while (current != null) {  if(element.compareTo(current.element) < 0) {  parent = current;  current = current.left;  //要插入的元素(5)比父节点(20)小，将其与该父节点的左节点(10)比较 ;发现比10还要小，继续想与10的左节点比较  //而10没有左节点，跳出循环 将该元素与10比较，小——创建节点  } else if(element.compareTo(current.element) > 0) {  parent = current;  current = current.right;  } else {  return false;  }  }    //插入  if(element.compareTo(parent.element) < 0) {  parent.left = createNewNode(element);  } else {  parent.right = createNewNode(element);  }    System.out.println(parent.toString());  }    return true;  }    // 创建新的节点  protected TreeNode<E> createNewNode(E e) {  return new TreeNode(e);  }  } |

上面代码展示了一个自己实现的简单的二叉查找树，包括了几个常见的操作。

因在二叉查找树中删除节点的操作比较复杂，下面详细介绍一下。

s

**3.4.4.2二叉查找树中删除节点分析**

要在二叉查找树中删除一个元素，首先需定位包含该元素的节点，及它的父节点。假设current指向二叉查找树中包含该元素的节点，而parent指向current节点的父节点，current节点可能是parent节点的左孩子或右孩子。

需考虑两种情况：

|  |
| --- |
| * 1.current节点没左孩子，只需将patent节点和current节点的右孩子相连 * 2. current节点有一个左孩子，假设rightMost指向包含current节点的左子树中最大元素的节点，而parentOfRightMost指向rightMost节点的父节点。那么先使用rightMost节点中的元素值替换current节点中的元素值，将parentOfRightMost节点和rightMost节点的左孩子相连，然后删除rightMost节点。   对2的总结：也就是从current节点的左子树中中找出最大的节点（该节点一定没有右孩子），替换current节点。 |

|  |
| --- |
| // 二叉搜索树删除节点  public boolean delete(E e) {  TreeNode<E> parent = null;  TreeNode<E> current = root;  // 找到要删除的节点的位置  while (current != null) {  if (e.compareTo(current.element) < 0) {  parent = current;  current = current.left;  } else if (e.compareTo(current.element) > 0) {  parent = current;  current = current.right;  } else {  break;  }  }  // 没找到要删除的节点  if (current == null) {  return false;  }  // 考虑第一种情况  if (current.left == null) {  if (parent == null) {  root = current.right;  } else {  if (e.compareTo(parent.element) < 0) {  parent.left = current.right;  } else {  parent.right = current.right;  }  }  } else { // 考虑第二种情况  TreeNode<E> parentOfRightMost = current;  TreeNode<E> rightMost = current.left;  // 找到左子树中最大的元素节点（该节点无右节点）  while (rightMost.right != null) {  parentOfRightMost = rightMost;  rightMost = rightMost.right;  }  // 替换  current.element = rightMost.element;  // parentOfRightMost和rightMost左孩子相连  if (parentOfRightMost.right == rightMost) {  parentOfRightMost.right = rightMost.left;  } else {  parentOfRightMost.left = rightMost.left;  }  }  return true;  } |

**3.5 平衡二叉树**

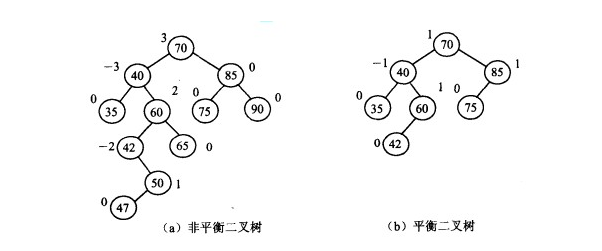
3.5.1 概述

又称AVL树

或是空树

或是具有下列性质的二叉树：

左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差绝对值不超过1。



3.5.2平衡二叉树

AVL树是最先发明的自平衡二叉查找树算法。

在AVL中任何节点的两个儿子子树的高度最大差别为1，也被称为高度平衡树，n个结点的AVL树最大深度约1.44log2n。

查找、插入和删除在平均和最坏情况下都是O（log n）。

增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

3.5.3红黑树

平衡二叉树的一种，保证最坏情况下基本动态集合操作的事件复杂度为O(log n)。

红黑树和平衡二叉树区别：

(1) 红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

(2) 平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

**四、图**

* **4.1 简介**

图是一种较线性表和树更为复杂的数据结构，在线性表中，数据元素之间仅有线性关系，在树形结构中，数据元素之间有着明显的层次关系，而在图形结构中，节点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。图的应用相当广泛，特别是近年来的迅速发展，已经渗入到诸如语言学、逻辑学、物理、化学、电讯工程、计算机科学以及数学的其他分支中。

**一、算法概述**

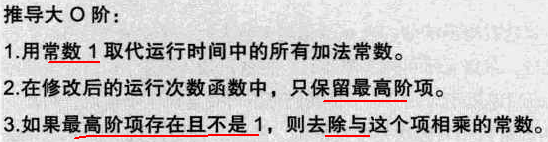
语言只是工具，算法才是程序设计的灵魂。

**1.1 算法简介**

算法是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。对于同一个问题的解决，可能会存在着不同的算法，为了衡量一个算法的优劣，提出了空间复杂度与时间复杂度这两个概念。



**1.2 时间复杂度**



一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数f(n)，算法的时间度量记为 **T(n) = O(f(n))** ，表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，简称时间复杂度。这里需要重点理解这个增长率。

举个例子，看下面3个代码：

|  |
| --- |
| 1、{++x;}  2、for(i = 1; i <= n; i++) { ++x; }  3、for(j = 1; j <= n; j++)  for(j = 1; j <= n; j++)  { ++x; } |

上述含有 ++x 操作的语句的频度分别为1 、n 、n^2，

假设问题的规模扩大了n倍，3个代码的增长率分别是1 、n 、n^2

时间复杂度分别为O(1)、O(n )、O(n^2)

**1.2.1 O(1): (常数阶)**

与问题大小无关(n的多少)，执行时间恒定。比如单纯的分支结构。

|  |
| --- |
| /\*\*  \* f(n)=3  \* 1.常数阶  \*/  public static void constantsOrder() {  int sum = 0, n= 100;  sum = (1 + n)\*n/2;  System.out.println("sum:" + sum);  } |

**1.2.2 O(n): (线性阶)**

|  |
| --- |
| /\*\*  \* f(n)=O(3)  \* 2.线性阶  \*/  public static void lineOrder(int n) {  for(int i=0;i<n;i++) {  //时间复杂度为O(1)的1程序步骤序列  }  } |

**1.2.3 O(n): (对数阶)**

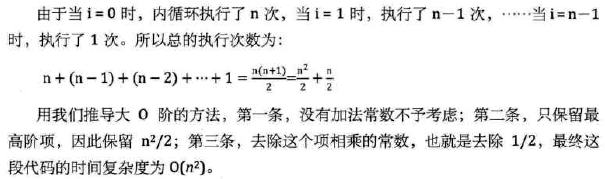
|  |
| --- |
| /\*\*  \* 3.对数阶  \* f(n)=O(logn)  \* 2^x>n时,会退出循环  \*/  public static void logOrder(int n) {  int count = 1;  while (count < n) {  count = count\*2;  //时间复杂度为O(1)的1程序步骤序列  }  } |

**1.2.4 O(n^2): 平方阶**

|  |
| --- |
| /\*\*  \* 4.平方阶  \* f(n)=O(n^2)  \*/  public static void squareOrder(int n) {  for(int i=0; i<n ;i++) {  for(int j=0;j<n;j++) {  //时间复杂度为O(1)的1程序步骤序列  }  }  } |

1.2.4.2 例子

|  |
| --- |
| /\*\*  \* f(n)=O(n^2)  \*/  public static void squareOrder2(int n) {  for(int i=0;i<n;i++) {  for(int j=i;j<n;j++) {  //时间复杂度为O(1)的1程序步骤序列  }  }  } |



1.2.4.3 例3: f(n)=O(n^2)

|  |
| --- |
| public static void example2(int n) {  n++; //执行次数1  function(0, n); //执行次数n  for(int i=0; i<n;i++){ //执行次数n^2  function(i, n);  }    for(int i =0; i<n;i++) { //执行次数n\*(n+1)/2  for(int j=i;j<n;j++) {  //时间复杂度为O(1)的1程序步骤序列  }  }  }    public static void function(int i, int n) {  for(int j=i; j<n; j++) {  //时间复杂度为O(1)的1程序步骤序列  }  } |



**1.2.5 常见时间复杂度**



**耗费的时间从小到大:**



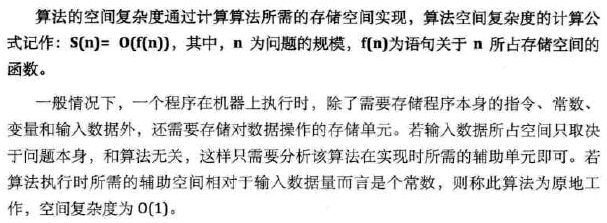
**1.2.6 最坏情况与平均情况**

没有特殊说明的话，都指的最坏时间复杂度

平均时间复杂度最有意义，是期望的运行时间

**1.3空间复杂度**

对一个算法在运行过程中临时占用存储空间大小的量度，记做S(n)=O(f(n))。



**二、查找算法**

查找和排序是最基础也最重要的两类算法，熟练掌握并能对其性能进行分析很重要，主要包括二分查找、快速排序、归并排序等等。

**2.1 顺序查找**

又称线性查找。

过程为：从查找表的最后一个元素开始逐个与给定关键字比较，若某个记录的关键字和给定值比较相等，则查找成功，否则，若直至第一个记录，其关键字和给定值比较都不等，则表明表中没有所查记录查找不成功，缺点是效率低下。

* 1. **二分查找**

**2.2.1 简介**

又称折半查找，对有序表来说，优点是比较次数少，查找速度快，平均性能好。

**基本思想**：将n个元素分成大致相等的两部分，取a[n/2]与x做比较，

如x=a[n/2],则找到x，算法中止；

如x<a[n/2]，则只要在数组a的左半部分继续搜索x，

如x>a[n/2]，则只要在数组a的右半部搜索x。

时间复杂度为O(logn)

**2.2.2 实现**

//给定有序查找表array 二分查找给定的值data

//查找成功返回下标 查找失败返回-1

|  |  |
| --- | --- |
| package search.method;  /\*\*  \* 二分查找  \*/   |  | | --- | | public class FunBinSearch {  public static int funBinsearch(int[] array, int data) {    int low = 0;  int high = array.length -1 ;    //while(array[low] < array[high]) {  while(low <= high) {  int mid = (low + high)/2;    if(data == array[mid]) {  return mid;  } else if(data < array[mid]) {  high = mid -1;  } else {  low = mid + 1;  }  }  return -1;  } |   public static void main(String[] args) {  int[] array = new int[] {1,2,4,5,7,9,11,13,16,19};  int result = funBinsearch(array, 10);  System.out.println("result:"+ result);  }  } |

**三、排序算法**

**3.1 分类**

功能是将一个数据元素(或记录)的任意序列,重排列成一个按关键字有序的序列。



**3.2 常见排序算法性能比较**



排序的稳定性：如果在排序的序列中，存在前后相同的两个元素的话，排序前和排序后他们的相对位置不发生变化。

* 1. **冒泡排序**

**3.3.1简介**

**基本思想**:设排序序列的记录个数为n，进行n-1次遍历，每次遍历从开始位置依次往后比较前后相邻元素，这样较大的元素往后移，n-1次遍历结束后,序列有序。

例如，对(3,2,1,5)排序过程：

共进行3次遍历

第1次遍历时先比较3和2，交换，继续比较3和1,交换，再比较3和5，不交换，这样第1次遍历结束，最大值5在最后的位置，得到序列(2,1,3,5)。

第2次遍历时先比较2和1，交换，继续比较2和3，不交换，第2次遍历结束时次大值3在倒数第2的位置，得到序列(1,2,3,5)

第3次遍历时，先比较1和2，不交换，得到最终有序序列(1,2,3,5)。

注意:如在某次遍历中没有发生交换，就不必进行下次遍历，因为序列已经有序。

**3.3.2实现**

|  |
| --- |
| // 冒泡排序 注意 flag 的作用  static void funBubbleSort(int[] array) {  boolean flag = true;  for (int i = 0; i < array.length - 1 && flag; i++) {//排的第i趟  flag = false;  for (int j = 0; j < array.length - 1 - i; j++) {//数组下标j  if (array[j] > array[j + 1]) {  int temp = array[j];  array[j] = array[j + 1];  array[j + 1] = temp;  flag = true;  }  }  }  for (int i = 0; i < array.length; i++) {  System.out.println(array[i]);  }  } |

**3.3.3 分析**

最佳情况：只需一次遍历就能确定数组已经排好序，不需要进行下一次遍历，最时，时间复杂度为 **O(n)** 。

最坏情况：需n-1次遍历，第一次遍历需要比较n-1次，第二次遍历需要n-2次，...，最后一次需要比较1次，最差情况下时间复杂度为 **O(n^2)** 。

* 1. **简单选择排序**

**3.4.1简介**

思想：设记录个数n，进行n-1次选择，第i次，在n-i+1(i = 1,2,...,n-1)个记录中选择关键字最小的记录作为有效序列中的第i个记录。

**3.4.2实现**

|  |
| --- |
| /\*\*  \* 思想:共排n-1次;第i次,从n-i个数中找到最小的，放在位置(i-1)上  \*/  static void funSelectionSort(int[] array) {  for (int i = 0; i < array.length - 1; i++) {  int mink = i;  // 每次从未排序数组中找到最小值的坐标  for (int j = i + 1; j < array.length; j++) {  if (array[j] < array[mink]) {  mink = j; //在第i次选择时，使mink指向最小的数对应的下标  }  }  // 将最小值放在最前面; i位置与最小值交换  if (mink != i) {  int temp = array[mink];  array[mink] = array[i];  array[i] = temp;  }  }  for (int i = 0; i < array.length; i++) {  System.out.print(array[i] + " ");  }  } |

测试结果:

|  |
| --- |
| 开始：{ 9,7,8,3,49,0,5,7,2}  第1次排后: {0, 7, 8, 3 , 49, 9, 5, 7, 2}  第2次排后：{0，2，8，3，49，9，5，7，7} |

**3.4.3分析**

需进行比较的次数与初始状态下待排序记录序列的排列情况**无关**。

当i=1时，需进行n-1次比较；

当i=2时，需进行n-2次比较；

依次类推，共需要进行的比较次数是(n-1)+(n-2)+…+2+1=n(n-1)/2，即进行比较操作的时间复杂度为 **O(n^2)** ，进行移动操作的时间复杂度为 **O(n)** 。

总的时间复杂度为 **O(n^2)** 。

最好情况下，即待排序记录初始状态就已经是正序排列了，则不需要移动记录。最坏情况下，即待排序记录初始状态是按第一条记录最大，之后的记录从小到大顺序排列，则需要移动记录的次数最多为3（n-1）。

简单选择排序是不稳定排序。

* 1. **直接插入排序**

**3.5.1简介**

思想：将一个记录插入到已排好序的有序表中,得到新的、记录数增1的有序表。

例如，排序序列(3,2,1,5):

初始有序序列为(3)

然后从位置1开始，先访问到2，将2插入到3前面，得到有序序列(2,3)

之后访问1,找到合适的插入位置后得到有序序列(1,2,3)

最后访问5，得到最终有序序列(1,2,3,5).

**3.5.2实现**

|  |
| --- |
| static void funDInsertSort(int[] array) {  int j;  for (int i = 1; i < array.length; i++) {  int temp = array[i];  j = i - 1;  //例:第2次,i=2,array[2] =1;将1与array[1]比较，若后者大，后者后移一位;再通过(j--)实现，array[0]与1比较，若前者大，前者后移一位。j-- 之后j=-1，没有数可以去与1比了，于是将1放在该位置（array[j + 1]）。  while (j > -1 && temp < array[j]) {  array[j + 1] = array[j];  j--;  }  array[j + 1] = temp;  }  } |

**3.5.3分析**

* 最好情况:待排序序列中记录已有序时，需n-1次比较，不需移动，时间复杂度为 **O(n)** 。
* 最差情况:当待排序序列中记录正好逆序时，则比较次数和移动次数都达到最大值，时间复杂度为 **O(n^2)** 。
* 平均情况下，时间复杂度为 **O(n^2)** 。
  1. **希尔排序**

又称“缩小增量排序”

是基于直接插入排序的以下两点性质而提出的一种改进：

(1) 直接插入排序在对几乎已经排好序的数据操作时，效率高，即可达到线性排序的效率。

(2) 直接插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。

* 1. **归并排序**

**3.7.1简介**

分治法的典型应用

思想：将待排序序列分为两部分，对每部分递归地应用归并排序，在两部分都排好序后进行合并。

例如，排序序列(3,2,8,6,7,9,1,5):

先将序列分为两部分:(3,2,8,6)和(7,9,1,5)

然后对两部分分别应用归并排序:

第1部分继续分为(3,2)和(8,6)，(3,2)继续分为(3)和(2)，(8,6)继续分为(8)和(6)，之后进行合并得到(2,3)，(6,8)，再合并得到(2,3,6,8)，

第2部分进行归并排序得到(1,5,7,9)，最后合并两部分得到(1,2,3,5,6,7,8,9)。

**3.7.2实现**

|  |
| --- |
| //归并排序  static void funMergeSort(int[] array) {  if (array.length > 1) {  int length1 = array.length / 2;  int[] array1 = new int[length1];  System.arraycopy(array, 0, array1, 0, length1);  funMergeSort(array1);  int length2 = array.length - length1;  int[] array2 = new int[length2];  System.arraycopy(array, length1, array2, 0, length2);  funMergeSort(array2);  int[] datas = merge(array1, array2);  System.arraycopy(datas, 0, array, 0, array.length);  }  } |

|  |
| --- |
| //合并两个数组  static int[] merge(int[] list1, int[] list2) {  int[] list3 = new int[list1.length + list2.length];  int count1 = 0;  int count2 = 0;  int count3 = 0;  while (count1 < list1.length && count2 < list2.length) {  if (list1[count1] < list2[count2]) {  list3[count3++] = list1[count1++];  } else {  list3[count3++] = list2[count2++];  }  }  while (count1 < list1.length) {  list3[count3++] = list1[count1++];  }  while (count2 < list2.length) {  list3[count3++] = list2[count2++];  }  return list3;  } |

**3.7.3分析**

归并排序的时间复杂度为O(nlogn)，它是一种稳定的排序，java.util.Arrays类中的sort方法就是使用归并排序的变体来实现的。

* 1. **快速排序**

**3.8.1简介**

思想：在待排序的序列中选择一个称为主元的元素，将数组分为两部分，使得第一部分中的所有元素都小于或等于主元，而第二部分中的所有元素都大于主元，然后对两部分递归地应用快速排序算法。

**3.8.2实现**

<http://www.cnblogs.com/coderising/p/5708801.html>

<http://www.cnblogs.com/hexiaochun/archive/2012/09/03/2668324.html>

**//快速排序**

**public class QuickSort {**

**//核心代码**

**/\*\***

**\* a[0]为基准点key(与第一个lo重合),lo-起始,hi-末尾**

**\* 从后半部分开始，发现有元素比key(第一个lo)小,就交换hi与lo对应的值(大的换到右边)**

**\* 接着从前半部分开始扫描(lo<key时,lo指针右移)，发现有元素大于key的值(始终是开始那个6)，就交换lo和hi的值**

**\* 直到lo>=hi,就把基准点放在hi的位置，一次排序完成**

**\* 以后采用递归分别对前半部分和后半部分排序**

**\*/**

**public void \_quick\_sort(int[] arrays, int start, int end) {**

**if(start>=end){**

**return;**

**}**

**int i = start;**

**int j = end;**

**int value = arrays[i];//基准点**

**boolean flag = true;**

**while (i != j) {//直到i=j，完成第一次排序**

**if (flag) {**

**if (value > arrays[j]) {//右边的hi值小于key时，交换lo和hi**

**swap(arrays, i, j);**

**flag=false;//交换值后，结束循环，转向左部分的判断**

**} else {**

**j--;//右边的hi值大于key时，hi指针左移**

**}**

**}else{**

**if(value<arrays[i]){**

**swap(arrays, i, j);**

**flag=true;**

**}else{**

**i++;**

**}**

**}**

**}**

**//此时,i=j,左右部分分别递归**

**\_quick\_sort(arrays, start, j-1);**

**\_quick\_sort(arrays, i+1, end);**

**}**

**//交换**

**private void swap(int[] arrays, int i, int j) {**

**int temp = arrays[i];**

**arrays[i] = arrays[j];**

**arrays[j] = temp;**

**}**

**public static void main(String args[]) {**

**QuickSort q = new QuickSort();**

**int[] a = { 49, 38, 65,12,45,5};**

**q.quick\_sort(a,6);**

**snp(a);**

**}**

**//判空**

**public void quick\_sort(int[] arrays, int lenght) {**

**if (null == arrays || lenght < 1) {**

**return;**

**}**

**\_quick\_sort(arrays, 0, lenght - 1);**

**}**

**//打印数组**

**public static void snp(int[] arrays) {**

**for (int i = 0; i < arrays.length; i++) {**

**System.out.print(arrays[i] + " ");**

**}**

**}**

**}**

|  |
| --- |
| // 快速排序  static void funQuickSort(int[] mdata, int start, int end) {  if (end > start) {  int pivotIndex = quickSortPartition(mdata, start, end);  funQuickSort(mdata, start, pivotIndex - 1);  funQuickSort(mdata, pivotIndex + 1, end);  }  } |

|  |
| --- |
| // 快速排序前的划分  static int quickSortPartition(int[] list, int first, int last) {  int pivot = list[first];  int low = first + 1;  int high = last;  while (high > low) {  while (low <= high && list[low] <= pivot) {  low++;  }  while (low <= high && list[high] > pivot) {  high--;  }  if (high > low) {  int temp = list[high];  list[high] = list[low];  list[low] = temp;  }  }  while (high > first && list[high] >= pivot) {  high--;  }  if (pivot > list[high]) {  list[first] = list[high];  list[high] = pivot;  return high;  } else {  return first;  }  } |

**3.8.3分析**

在快速排序算法中，比较关键的一个部分是主元的选择。

* 在最差情况下，划分由n个元素构成的数组需要进行n次比较和n次移动，因此划分需要的时间是O(n)。
* 在最差情况下，每次主元会将数组划分为一个大的子数组和一个空数组，这个大的子数组的规模是在上次划分的子数组的规模上减1，这样在最差情况下算法需要(n-1)+(n-2)+...+1= **O(n^2)** 时间。
* 最佳情况下，每次主元将数组划分为规模大致相等的两部分，时间复杂度为 **O(nlogn)** 。
  1. **堆排序**

**3.9.1简介**

n个关键字序列K1，K2，…，Kn称为堆，

当且仅当该序列满足如下性质（简称为堆性质）：

1. ki <= k(2i）且 ki <= k(2i+1) (1 ≤ i≤ n/2）:小根堆
2. 大根堆则换成>=号。

如果将上面满足堆性质的序列看成是一个完全二叉树，则堆的含义表明，完全二叉树中所有的非终端节点的值均不大于（或不小于）其左右孩子节点的值。

意思是：给定一个待排序序列，首先经过一次调整，将序列构建成一个大顶堆，此时第一个元素是最大的元素，将其和序列的最后一个元素交换，然后对前n-1个元素调整为大顶堆，再将其第一个元素和末尾元素交换，这样最后即可得到有序序列。

**3.9.2实现**

**http://www.cnblogs.com/CherishFX/p/4643940.html#3661395**

|  |
| --- |
| //堆排序  public class TestHeapSort {  public static void main(String[] args) {  int arr[] = { 5, 6, 1, 0, 2, 9 };  heapsort(arr, 6);  System.out.println(Arrays.toString(arr));  }  static void heapsort(int arr[], int n) {  // 先建大顶堆  for (int i = n / 2 - 1; i >= 0; i--) {  heapAdjust(arr, i, n);  }  for (int i = 0; i < n - 1; i++) {  swap(arr, 0, n - i - 1);  heapAdjust(arr, 0, n - i - 1);  }  }  // 交换两个数  static void swap(int arr[], int low, int high) {  int temp = arr[low];  arr[low] = arr[high];  arr[high] = temp;  } |

|  |
| --- |
| // 调整堆  static void heapAdjust(int arr[], int index, int n) {  int temp = arr[index];  int child = 0;  while (index \* 2 + 1 < n) {  child = index \* 2 + 1;  // child为左右孩子中较大的那个  if (child != n - 1 && arr[child] < arr[child + 1]) {  child++;  }  // 如果指定节点大于较大的孩子 不需要调整  if (temp > arr[child]) {  break;  } else {  // 否则继续往下判断孩子的孩子 直到找到合适的位置  arr[index] = arr[child];  index = child;  }  }  arr[index] = temp;  }  } |

package sort.method;

public class HeapSort {

//堆排序

public static int[] heapSort(int[] array){

array = buildMaxHeap(array); //初始建堆，array[0]为第一趟值最大的元素

for(int i=array.length-1;i>=1;i--){

int temp = array[0];

//将堆顶元素和堆低元素交换，即得到当前最大元素正确排序位置

array[0] = array[i];

array[i] = temp;

adjustDownToUp(array, 0,i); //整理，将剩余的元素整理成堆

}

return array;

}

//构建大根堆：将array看成完全二叉树的顺序存储结构

private static int[] buildMaxHeap(int[] array){

//从最后一个节点array.length-1的父节点（array.length-1-1）/2开始，

直到根节点0，反复调整堆

for(int i=(array.length-2)/2;i>=0;i--){

adjustDownToUp(array, i,array.length);

}

return array;

}

//将元素array[k]自下往上逐步调整树形结构

private static void adjustDownToUp(int[] array,int k,int length){

int temp = array[k];

for(int i=2\*k+1; i<=length-1; i=2\*i+1){

//i为初始化为节点k的左孩子，沿节点较大的子节点向下调整

if(i+1<length && array[i]<array[i+1]){

//取节点较大的子节点的下标

i++; //如果节点的右孩子>左孩子,则取右孩子节点的下标

}

if(temp>=array[i]){

//根节点 >=左右子女中关键字较大者，调整结束

break;

}else{ //根节点 <左右子女中关键字较大者

array[k] = array[i];

//将左右子结点中较大值array[i]调整到双亲节点上

k = i; //【关键】修改k值，以便继续向下调整

}

}

array[k] = temp; //被调整的结点的值放人最终位置

}

//http://www.cnblogs.com/CherishFX/p/4643940.html#3661395

public static void main(String[] args) {

int[] a = {14,12,15,13,11,16};

printArray("排序前：",a);

heapSort(a);

printArray("排序后：",a);

}

private static void printArray(String pre,int[] a) {

System.out.print(pre+"\n");

for(int i=0;i<a.length;i++)

System.out.print(a[i]+"\t");

System.out.println();

}

}

**3.9.3分析**

由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

时间复杂度也为O(nlogn)，空间复杂度为O(1)。

不稳定的排序方法。

与快排和归并排序相比，堆排序在最差情况下的时间复杂度优于快排，空间效率高于归并排序。

**四、其它算法**

尤其是二分查找、快排、堆排、归并排序这几个更是面试高频考察点。

递归

* **简介**

在平常解决一些编程或者做一些算法题的时候，经常会用到递归。程序调用自身的编程技巧称为递归。它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。上面介绍的快速排序和归并排序都用到了递归的思想。

* **经典例子**

斐波那契数列，又称黄金分割数列、因数学家列昂纳多·斐波那契以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：0、1、1、2、3、5、8、13、21、34、……在数学上，斐波纳契数列以如下被以递归的方法定义：F（0）=0，F（1）=1，F（n）=F(n-1)+F(n-2)（n≥2，n∈N\*）。

//斐波那契数列 递归实现

static long funFib(long index) {

if (index == 0) {

return 0;

} else if (index == 1) {

return 1;

} else {

return funFib(index - 1) + funFib(index - 2);

}

}

上面代码是斐波那契数列的递归实现，然而我们不难得到它的时间复杂度是O(2^n)，递归有时候可以很方便地解决一些问题，但是它也会带来一些效率上的问题。下面的代码是求斐波那契数列的另一种方式，效率比递归方法的效率高。

static long funFib2(long index) {

long f0 = 0;

long f1 = 1;

long f2 = 1;

if (index == 0) {

return f0;

} else if (index == 1) {

return f1;

} else if (index == 2) {

return f2;

}

for (int i = 3; i <= index; i++) {

f0 = f1;

f1 = f2;

f2 = f0 + f1;

}

return f2;

}

分治算法

分治算法的思想是将待解决的问题分解为几个规模较小但类似于原问题的子问题，递归地求解这些子问题，然后合并这些子问题的解来建立最终的解。分治算法中关键地一步其实就是递归地求解子问题。关于分治算法的一个典型例子就是上面介绍的归并排序。[查看更多关于分治算法的内容](http://baike.baidu.com/link?url=5po48WW5iqSwgWe3Agx1J3O7tsbmVgLMK_B_QshUsABVsGacspnKUQZdar-YZee08SSiCVbUPbvJAw3wGByJR-IGXR7Hw8Awmiwj7AxF5GeR2aFVWwNHbfXBB5OzZw--)

动态规划

动态规划与分治方法相似，都是通过组合子问题的解来求解待解决的问题。但是，分治算法将问题划分为互不相交的子问题，递归地求解子问题，再将它们的解组合起来，而动态规划应用于子问题重叠的情况，即不同的子问题具有公共的子子问题。动态规划方法通常用来求解最优化问题。[查看更多关于动态规划的内容](http://baike.baidu.com/link?url=CCSb2DEtf8-ATMN3WE9dq_6JHrxtKbIlftZxnFmECp_-dS8ZprtbPsiH2wkn-LT5lPeW8gB4Pq-7InGtksB3lMvB82_LJn2DeeEtK1s3RcUu80YAxjc3N8rBXW9eNlTY)

动态规划典型的一个例子是[最长公共子序列](http://www.cnblogs.com/huangxincheng/archive/2012/11/11/2764625.html)问题。

常见的算法还有很多，比如贪心算法，回溯算法等等，这里都不再详细介绍，想要熟练掌握，还是要靠刷题，刷题，刷题，然后总结。

**五、常见算法题**

下面是一些常见的算法题汇总。

不使用临时变量交换两个数

static void funSwapTwo(int a, int b) {

a = a ^ b;

b = b ^ a;

a = a ^ b;

System.out.println(a + " " + b);

}

判断一个数是否为素数

static boolean funIsPrime(int m) {

boolean flag = true;

if (m == 1) {

flag = false;

} else {

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

if (m % i == 0) {

flag = false;

break;

}

}

}

return flag;

}

其它算法题

1、[15道使用频率极高的基础算法题](http://www.codeceo.com/article/15-algorithms-question.html)  
2、[二叉树相关算法题](http://blog.csdn.net/luckyxiaoqiang/article/details/7518888/)  
3、[链表相关算法题](http://blog.csdn.net/kerryfish/article/details/24043099)  
4、[字符串相关算法问题](http://blog.chinaunix.net/uid-20498361-id-1940276.html)