首先要根据程序要处理的数据（包括输入的数据和输出的数据）设计数据结构，再设计算法来实现程序要达到的功能。最后才使用一门编程语言来编码。

数据结构>算法>程序设计语言。

数据结构是算法实现的基础。

分治算法：对于一个规模为N的问题，若该问题可以分解为M个规模较小的问题（任何问题不都是由一个个小小的问题组成，这不是废话吗。。。看看是怎么work的吧）。这些子问题互相独立，并且和原来问题形式相同，递归解决这些子问题，然后将各个问题的解合并得到原来的解。------what？用到递归？

分治和递归算法经常同时应用到算法设计之中，并且可以由此产生许多高效的算法。

一般的，通过自动产生比赛赛程的例子，总结了一下，在进行分治的时候一般会碰到一个主要的问题就是，怎样将得到的结果在合并在一起得到最终的结果呢，这需要从得到的分治结果中去找到规律。这些子问题互相独立，并且和原来问题形式相同。

贪婪算法：从问题的某一初始解出发，追捕逼近给定的目标~~~不是人话，听不懂，或者说又是废话，非常生动，但是确实让人一下子脑子里啥都想不起来。

不过有个比较好的例子：超市给你找钱的时候，比如17.5，总是先去装10块的那个抽屉里去，再去装5块的抽屉里去取，最后去1块5毛的抽屉中取。

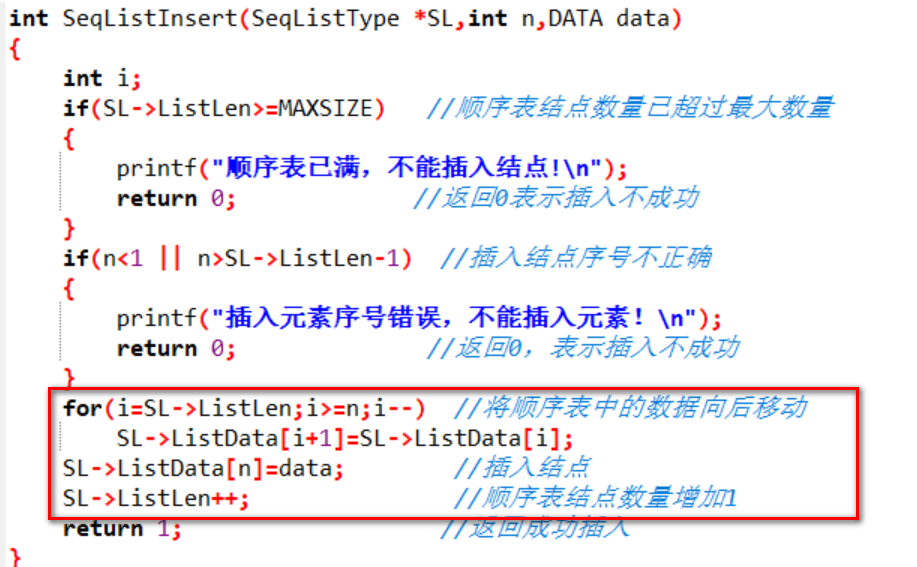
回溯算法（试探性算法）：以下棋为例，首先试想下一步的位置，以及估计对手的走位，再计算自己的走位，如果对手的走位对于自己不利，则取消下一步的设想-----这样应该是根据未来的情况决定当前的决定是否有利。

试探法是计算机解题中常见的算法，很多问题无法根据某种确定的计算法则来求解，可以先选择一种可能的情况，在试探的过程中，一旦发现原来选择的假设情况是错误的，就退回一步重新选择。

试探算法一般也采用递归的形式编写程序。

第二章 简单数据结构

顺序表示将数据按顺序保存在内存中，因此在顺序表中存取数据都很方便，但是可以看到也有一个缺点就是，在想顺序表中插入数据的时候，需要移动大量的数据，特别是如果我每次都是在顺序表的前面的位置插入，那么就要把之后的数据全部后移，这样效率非诚低，也没这个必要，容易出错：



一般，对于需要经常插入数据的线性表，采用链式存储的链表，这是一种动态存储分配方案，可以更具需要来申请内存单元，具体如何实现的，带着微信的态度往下看把

！！！！！！！！！！！！

链表中每个节点分为两部分，一部分是实际的数据，另外一部分是下个节点的地址，这样的话整个链表的存储就不是连续的了，要是插入数据的话只要修改两处地址就可以了。

链表分为两种：单向链表，就是上面介绍的；双向链表，一个节点里有2个指针，分别指向上一个节点和下一个节点。

在使用链表的时候，需要用到malloc函数动态分配节点的存储空间，而且当删除某个节点的时候要通过free函数释放占用的空间。

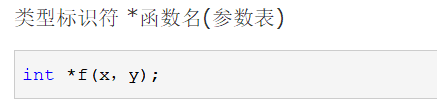
对链表的访问只能从表头开始访问，才能找到下一个节点的位置，但是顺序表的话就可以实现随机访问，能够快速的找到任何一个节点的位置。

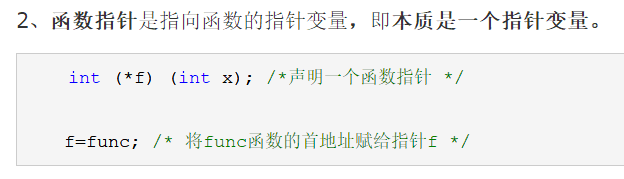
综上可以发现，顺序表查表快，对表的增减不合理；链表对节点的增减合理，但是查表节点慢。

Typedef: typedef是在计算机编程语言中用来为复杂的声明定义简单的别名

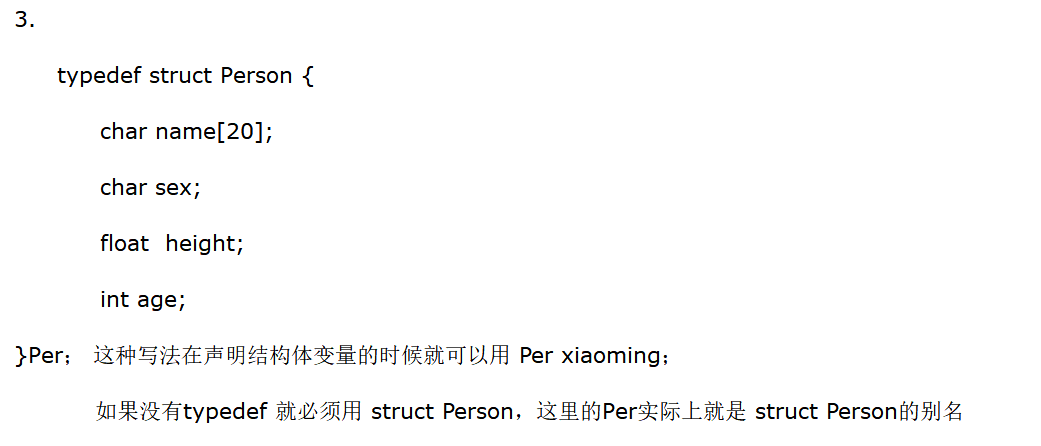
例子：typedef int INT;这个例子其实比较有误导性，好像把int换了个名字似的。但实际上，看作int INT（声明了一个整形变量），再在前面加上typedef 把INT转变为int的代名词.

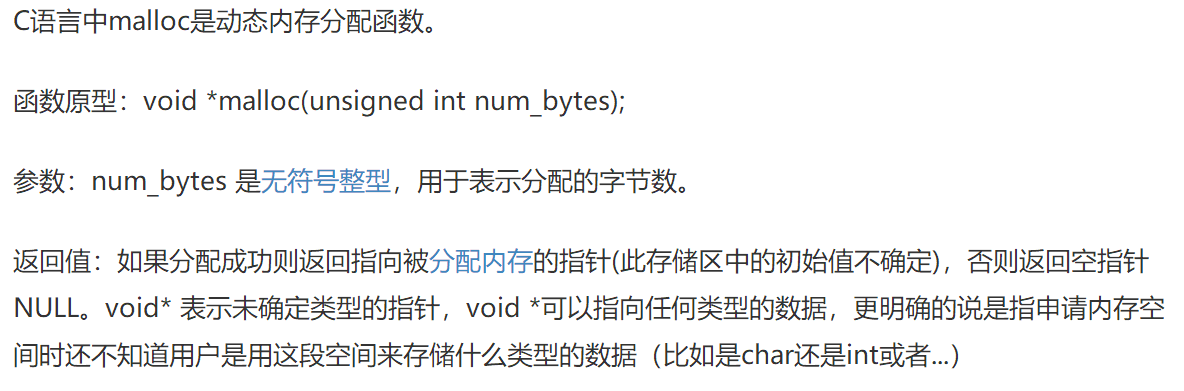
指针函数是指带指针的函数，即本质是一个函数，函数返回类型是某一类型的指针。











## 队列：先进先出

队列是一种特殊的线性表,只能对队首队尾操作: 主要有几种操作:

初始化队列,进队,出队,获取队列长度.

### 假溢出

先进行出队操作,head就会不断后移,如果tail已经指向了队列的最后,这时候如果进行入队的操作, 则仍然会提示队列已满.

解决方法?

在出队完成的时候,将队列后面的元素向前移动,占据刚才已经移除的元素的位置,这样就保证了队列的第一个位置有元素,但是这样的话效率特别低.

#### 另一种方法,就是队列的首尾相联,形成环形队列

Tail=(tail+1)%n; //n表示队列最大元素数

## 栈：后进先出（从栈顶出）

计算机程序设计中，有许多地方都需要使用栈，在调用函数的时候，就需要把函数的返回地址，主调函数各个变量等值入栈，等到调用函数执行完成之后，再从栈中弹出变量一遍回复主调函数的继续执行。

## 第三章 复杂的数据结构

## 树

对于之前的线性表，队列，栈，都是线性的，使用这种结构不能反映元素之间的层次特征，比如，国家的行政机构，在软件工程中，有很多地方采用树型层次结构来表示数据间的关系。

#### 树vs线性链表

线性链表一个节点有一个前驱节点和一个后继节点

树，一个前驱节点和多个后继节点

节点的度：一个节点的子数的数量称为该节点的度

树的度：该树中节点的最大度数

叶子节点：树中度为0的节点称为叶子节点，度不为0的节点称为分支节点

节点的层数：树是一种层次结构

树的深度：一棵树中节点的最大层数称为树的深度

有序树和无序树：树中各节点的子树按一定的顺序从左向右安排称为有序树

森林：m颗互不相交的树的集合

### 二叉树

二叉树要么为空，要么就只有根节点，或仅有左树，要么仅有右树，或者有左右子树

#### 树vs二叉树

树的节点的度数没有限制，二叉树节点最大度数为2

树的节点无左右之分，二叉树节点有左右之分

#### 满二叉树和完全二叉树

满：除了最后一层叶节点，其他节点都有2个子节点

完全：最后一层从左向右的叶节点连续存在，只缺少右侧若干节点

## 二叉树的性质

1. 二叉树中，第i层节点总数最多为2i-1个
2. 深度为k的二叉树最多2k-1个节点，最少有k个节点，左树
3. 一颗二叉树，叶节点n0,度为2的节点总数n2,则n0=n2+1;
4. 具有n个节点的完全二叉树的深度为k, k>=log2n+1
5. 对于一个具有n个节点的完全二叉树各个节点如果采用顺序方式存储，对于任意节点i，有如下关系：
6. 如果i!=1,则其父节点的编号为i/2；
7. 如果2i<=n，则其左子树的根节点编号为2i，如果2\*i>n，那么无左子树

因为对于完全二叉树，左子树的根节点编号总是2的倍数

1. 如果2\*i+1<=n,那么右子树根节点编号为2\*i+1;否则无右子树

## 二叉树顺序存储

对于完全二叉树，采用顺序存储，各节点之间具有对于的关系，对于节点i，其父节点的编号为i/2，其左子节点编号2\*i

若为非完全热二叉树，只能采用插空的方法，这样节点和序号没有对应关系，而且用15个接节点的存储空间来存储实际上只有7个节点，造成了存储空间的浪费

## 链式存储结构

##### 二叉链式结构vs三叉链式结构

一个节点由节点元素和两个分别指向左右子树的指针组成

三叉就增加了一个指向父节点的指针方便查找父节点

具有n个节点的二叉树，有2n个指针域，只有n-1个用来指向节点的左右子树，其他n+1指针域为空

## 遍历二叉树

先序遍历DLR：先访问根节点，再先序遍历左子树，最后先序遍历右子树

中序遍历LDR：先中序遍历左子树，再访问根节点，最后中序访问右子树

后序遍历LRD：先后序遍历左子树，再后序遍历右子树，最后访问根节点

## 线索二叉树

二叉链式结构只能找到节点左右子树的信息，而不能得到节点的前驱和后驱的信息，线索树就是：

首先可以想到为每一个二叉树节点再增加两个指针分别指向前驱和后驱节点，但是这样一来就增加了二叉树的存储开销。

其实，对于一个具有N个节点的二叉树来说，其具有2n个指针域，其中n-1个用于指向除根节点的n-1个节点，n+1个节点为空\*（证明？？？？？？？？？？）

所以有这样的思路就是：

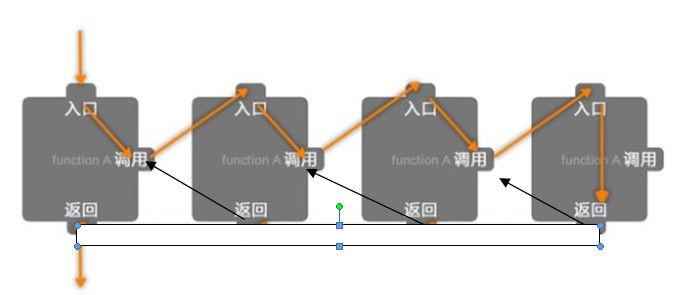
将那些指向空的指针域用来存放节点的前驱和后驱，这种在节点的空指针域中存放该节点的前驱和后驱节点的指针成为线索。

线索二叉树又可以分为先序线索二叉树，中序和后序线索二叉树。

在一个线索二叉树中，为了区别每个节点的左右指针域所存放的是子树指针还是线索，必须在节点结构中增加两个标志语lflag和rflag，若某个标志为1， 表示对应的指针域为线索，否则为子树指针。

左值向前驱，右指向后继

## 递归



## 最优二叉树（哈夫曼树）

哈夫曼树值的是一组具有确定权值的叶子节点的具有最小带权路径长度的二叉树（不懂。。。。。。。。。。。。。。。。。）

几个概念:

1. 节点的权： 根据实际应用中各个节点数据的重要性，赋予各个节点一个数，数值最大的其重要性最大，这个数值成为节点的权
2. 路径： 树中从一个节点到另一个节点之间的分支，称为路径
3. 路径长度： 一个路径上经过的分支的数量
4. 树的路径长度：从根节点到每个节点的路径长度之和，在节点数目相同的二叉树中，完全二叉树的路径长度最短
5. 节点的带权路径长度：若节点是带权的，就是该节点到根节点的路径长度与该节点的权的长度
6. 树的带权路径长度：所有节点的带权路径长度之和WPL

构建哈夫曼树

首先更具n个带权值的节点构建n个二叉树，每一个节点作为一个二叉树，其权值保存在节点中，左右子树都是空。

在这n个二叉树中找到权值最小的两棵树，用这两课树作为左右子树构造新的二叉树，将原来两个数的权值相加作为新的二叉树根节点的权值

将上步找到的权值最小的两颗二叉树排除在下一次查找范围，重复2,3直到只有一颗树为止，这样就可以得到一颗哈夫曼树

## 哈夫曼树的应用：哈夫曼编码

在通信，数据存储和数据传输时，一般要对需要处理的字符进行二进制编码，例如ASCII码就是一种编码方案，对于ASCII码，每个字符使用7位二进制进行编码，这种方式成为等长编码。

但是，实际上，，总是希望用较少的编码位保存或传输相同的信息，显然用等长编码不太合适，不等长编码就是按照字符出现的频率对不同字符长度使用不同长度的编码，对于使用频率高的字符使用短的编码，这样就缩短了整个信息的编码位数，减少码长。

译码二义性

在对编码进行译码的时候，选择第一个字符进行译码还是选择前两个进行译码这件事存在分歧，所以就要保证任意字符的编码都不是其他字符编码的前缀，这样的编码就成为前缀编码。

使用哈夫曼树就可以解决这个问题，因为在树中，任意叶子节点都不是其他叶子节点的前缀