针对研究生考试大纲设计

计算机考研之
数据结构高分笔记

——站在学生的角度辅导你考研

率辉 编著　　周伟 张浩 审核

立志于打造最贴近考生实际的辅导书

计算机考研之数据结构高分笔记

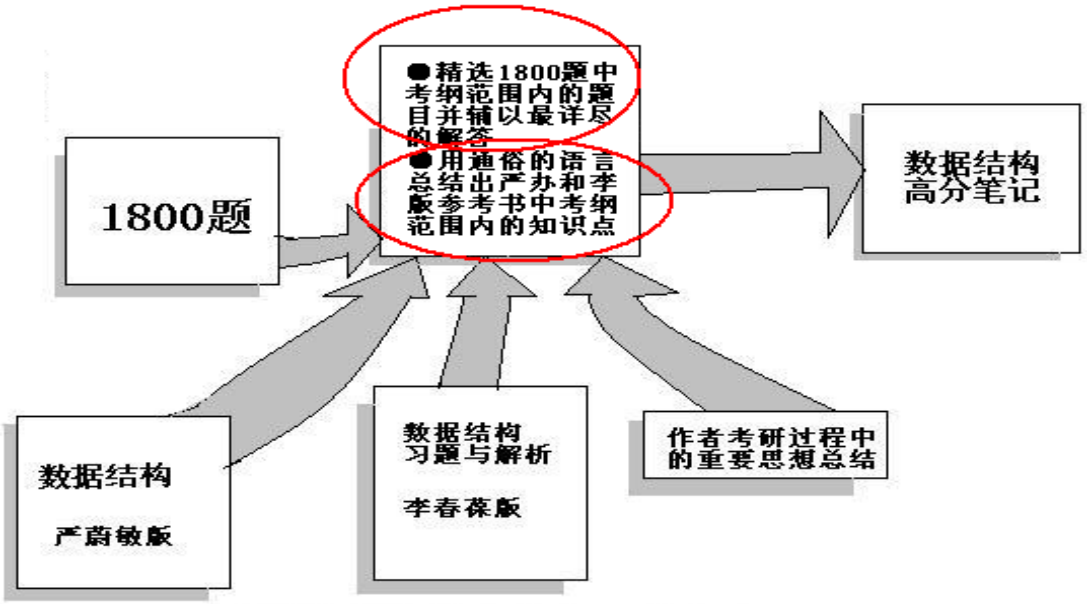
率辉 编著

周伟 张浩 审核

前言

在计算机统考的四门专业课中，最难拿高分的就是数据结构。但是这门课本身的难度并不是考生最大的障碍，真正的障碍在于考生不能独自把握复习的方向和考试范围。也许有同学要问了，我们不是有大纲吗？照着大纲去复习不就可以了吗？表面上看是这样，但是当你真正开始复习的时候你就会发现，其实大纲只给了考生一个大致范围，有很多地方是模糊的，这些模糊的地方就可能是你要纠结的地方。比如大纲里对于栈和队列的考查中有这么一条：“栈和队列的应用”。这个知识点就说的很模糊，因为只要涉及到栈和队列的地方，都是其应用的范畴，这时考生该怎么办呢？于是把所有的希望寄托于参考书，希望参考书能帮助我们理解大纲的意图。下边我们就来说说参考书吧。参考书分两种，一是课本，二是与课本配套的辅导书。对于课本，考生用的最多的就是严版的《数据结构》，这里我也推荐大家把这本书选作考研辅导教材。因为这本书的内容非常丰富，如果能把这本书中考纲要求的章节理解透彻了，参加考研没有任何问题。但是这个过程是漫长的，除非本科阶段就学的非常好。计算机统考后，专业课四门加上公共课三门，一共是七门。绝大数考生复习的时间一般也就六个月，而数据结构的复习需要占用多少时间，我不算大家也清楚。要在这么短的时间内掌握严版数据结构上考纲要求的知识点，基本上是不可能的，这就需要一本辅导书来依照大纲从课本中总结出考纲要求的知识点，才能使得考生在短时间内达到研究生考试的要求。市面上的参考书有两种，一种是四合一的辅导书，比如大家熟悉的复旦版的，山东人民出版社出版的等等。另一种是分册的，比如网上流行的《1800 题》以及其第二版，此书题目巨多，并且有很多老式的考研题，有些算法设计题的答案是 Pascal 语言写的。这本书中的题目一般考生全做基本上是不可能的，挑着做又会把时间浪费在选题上。不可否认，这本书确实是一本非常好的题库，但是考生直接拿来用做考研辅导书却不太合适。还有一本书叫《数据结构习题与解析》，作者是李春葆，上边总结了一些考研所需知识点，但是这本书同样出自统考以前，也不完全适应新大纲的要求。直到复试后，第一次见到周伟写的计算机网络高分笔记样稿，经过半天的研读，发现这个就是我想要的，如果这种写作风格用在数据结构我相信一定是最畅销的书籍。辅导书就应该站在学生的角度去写，特别像数据结构这门比较难深入的课。所以我决定加入周伟的队伍一起来完成一本真正意义站在学生角度去写的数据结构辅导书。去更好的帮助考生在最短的时间内去掌握这一门课。这也是我写这本辅导书最主要的动机。

接下来我详细讲解一下这本辅导书书的写作过程，请看下图：

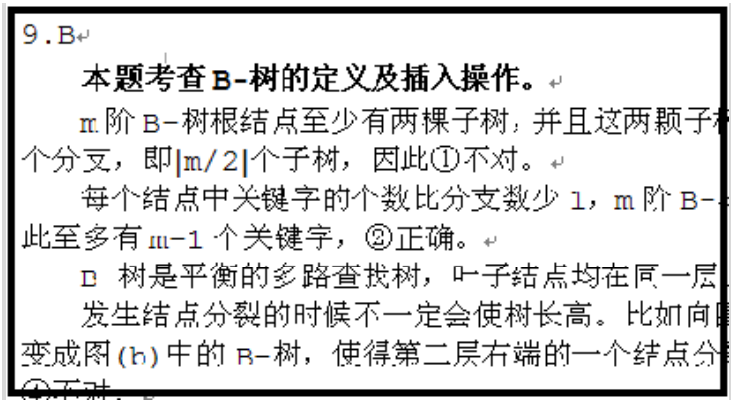


《数据结构高分笔记》的由来

图中所涉及的书都是大家很熟悉的吧。当年这些书我都买了，花了很大心思才从中找出在考研战场上真正有用的东西。比如《1800 题》，里边有好题，有废题，我当时多么渴望有人能在我复习之前就帮我从中去掉重复的题目，选出大纲要求的题目，并能把解答写的更通俗易懂点，可是当时没有人这么做，。而我们所做的工作就是从这 1800 题中选出了大纲要求的题目，并且修正了部分解答，使其更容易理解，我想这也是你们很想要的吧。其次是严版的《数据结构》写的过于严谨，语言表述过于专业，对于基础稍差的同学来说读起来十分费力，要很长时间才能适应这本书的写作风格。我当时就是在这本书中痛苦的挣扎了很久，看第三遍的时候才真正的可以说适应了，何苦这样呢？如果当时有一本辅导书帮我把那些复杂程序的执行过程，拗口的专业术语，令人头大的符号，翻译成容易理解的语言，我就可以节省很多时间，可惜当时也没有。而我们所做的就是根据自己复习的经验，以及对这本书的理解，把其中考试不需要的内容删掉，把需要的内容经过改造变成一般学生容易接受的形式。对于李版的《习题与解析》我也做了类似的处理。并且，我在本书中穿插讲解了一些考纲中没有明文规定但是很多算法题目中大量用到得算法设计课程中的思想，来帮助大家提高解算法设计题的能力，比如搜索（打印图中两节点之间的所有路径），分治法（二分法排序、求树的深度等等）等算法思想。因此我相信这本书会给你的考研复习带来很大的帮助。

本书特点：

（1）精心挑选出适合考研的习题，并配上通俗易懂的答案供你自测和练习。


9.B

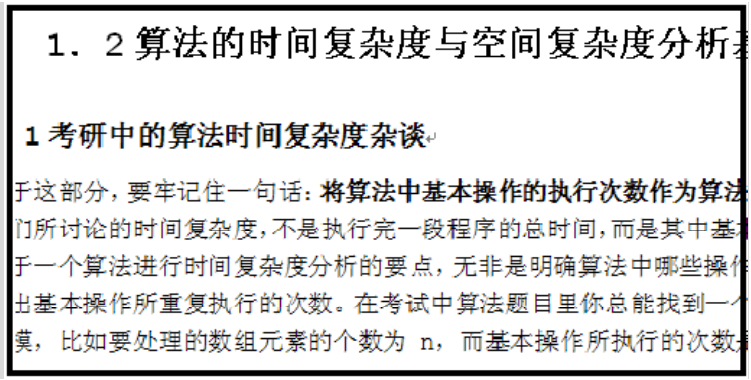
本题考查B-树的定义及插入操作。

m阶B-树根结点至少有两棵子树，并且这两颗子
个分叉，即⌈m/2⌉个子树，因此①不对。

每个结点中关键字的个数比分支数少1，m阶B-
此至多有m-1个关键字，②正确。

B 树是平衡的多路查找树，叶子结点均在同一层
发生结点分裂的时候不一定会使树长高。比如向
变成图(b)中的B-树，使得第二层右端的一个结点分
④不对。

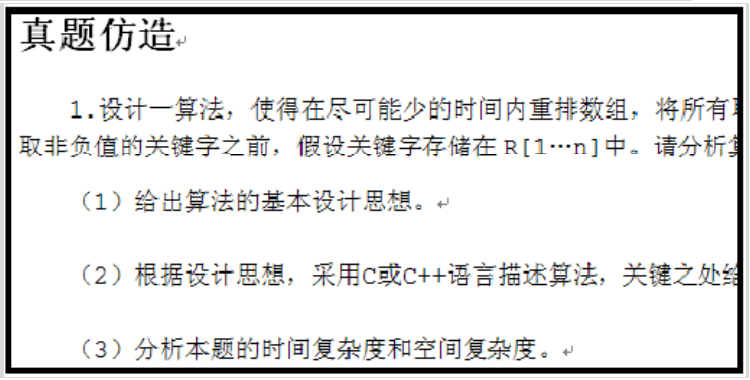
（2）总结出考研必备知识点，并且帮你把其中过于专业过于严谨的表述翻译成通俗易懂的语言。


1．2 算法的时间复杂度与空间复杂度分析

1 考研中的算法时间复杂度杂谈

于这部分，要牢记住一句话：**将算法中基本操作的执行次数作为算法**
们所讨论的时间复杂度，不是执行完一段程序的总时间，而是其中基
于一个算法进行时间复杂度分析的要点，无非是明确算法中哪些操
出基本操作所重复执行的次数。在考试中算法题目里你总能找到一
莫，比如要处理的数组元素的个数为 n，而基本操作所执行的次数

（3）针对于近年数据结构大题的出题风格（比如算法设计题目中的三段式题目：1.表述算法思想。2.写出算法描述。3.计算算法的时间和空间复杂度），设计了独特的真题仿造部分，让你在复习的过程中逐渐养成适合解决考研类型题目的习惯。


真题仿造

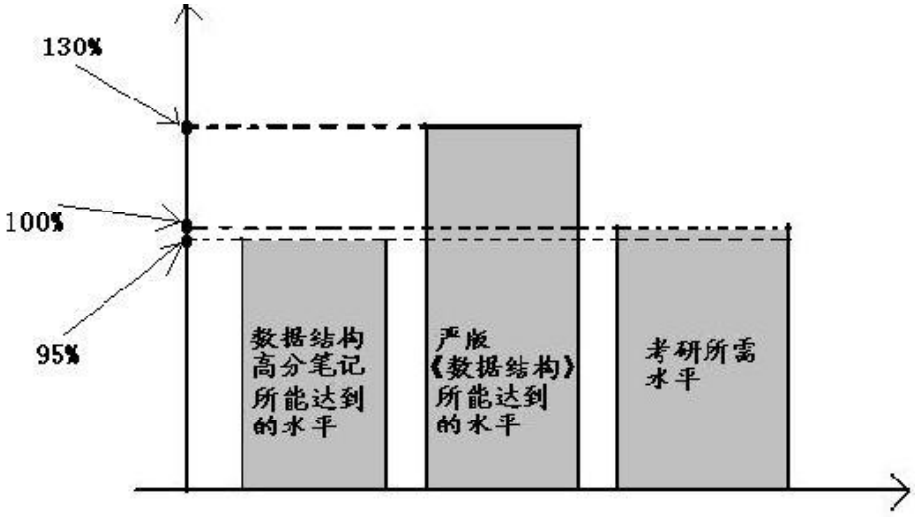
1.设计一算法，使得在尽可能少的时间内重排数组，将所有
取非负值的关键字之前，假设关键字存储在R[1…n]中。请分析

（1）给出算法的基本设计思想。

（2）根据设计思想，采用C或C++语言描述算法，关键之处

（3）分析本题的时间复杂度和空间复杂度。

听我说了这么多之后，很多学生现在想问，我只看你这本书够不够？还需要自己准备其他书吗？对于这个问题，我用下图来回答。



从图中可以看到，如果你只看本书，你能达到考研要求水平的 95%左右，为什么是这样，因为今年大纲还没有公布，所以我不敢保证我的书涵盖大纲所有内容。但是数据结构中的经典内容本书已经全部包括，再加上对统考这两年大纲范围的解读，估计今年大纲变化不会太大，毕竟数据结构是一门经典科目，因此考研对这一门科的考察范围较为稳定。从图中同样可以看出，掌握了严版《数据结构》你可以至少掌握比考试范围多出 30%的内容，但是这需要花很多时间，并不可行。因此在这里我建议大家先看本书，把重要知识点先拿到手，然后把严版数据结构当做字典来用，等正式大纲出来之后进行查缺补漏，这是一种较为高效的复习方法。这本书不仅涵盖了考纲绝大部分内容，更重要的是它会帮助你理解大纲，理解出题人的思路，这样你就会白哪一类的题目有可能考，哪一类的题目不会考，慢慢的，你复习的方向感会越来越明确，效率会越来越高。

本书作者

第一章 绪论

作者的话:

大部分同学在学习数据结构时，想必对数据结构课本里的伪代码多多少少有点不是很清楚，特别是自己在动手编写算法的时候，明明知道算法的思路，但是编写出来的程序就是不标准，可能在考试的时候就会吃大亏。所以在开始数据结构的旅程之前，我觉得有必要将一些基本功提前告知你们，掌握了这些东西，在本章以后的章节中，才能以此为基础来修炼更高深的武功。

本章概略

▲ 针对考研数据结构的 C&C++语言基础以及代码书写规范

对于考研数据结构，需要 C 与 C++语言作为基础，但是又不需要太多，因此此处讲解有针对性。现在我们面临的是研究生考试，要在答题纸上写代码，代码的评判者是阅卷老师，而不是 TC，VC6.0 等编译器。如果之前你只熟悉在这些编译器下写代码，那你要看看这一部分，这里教你怎么快速的写出能让阅卷老师满意的代码。

▲ 算法的时间复杂度分析基础

为什么要特别注重这一块的讲解？在 09 年批阅数据结构算法那道题的时候，由于当时阅卷的标准答案是教育部给出的，并且明确说明以此为标准答案，但是教育部给出的算法时间复杂度太大，对于算法有研究的同学，可以很轻松的写出一个算法，并且时间复杂度远远小于标准答案。教育部就是权威，没有办法，只能按照教育部的答案改，这样就导致了算法牛人写出更完美的算法，却得了最低的分。也许是为了避免这种不公平的再次出现，10 年的考试要求终于改了，考生必须对自己写的算法给出时间复杂度和空间复杂度，并以此来作为评分的依据。所以这已经成为数据结构 45 分里面的必考内容，**这一点的考察在图、排序、查找这三章内体现的尤为明显**，因此我会在本章先总体讲一下算法时间复杂度分析的基本方法，并在以后章节中以题目的形式讲一些具体分析思路，这样考生逐渐的就会掌握考研要求的算法复杂度分析方法。

▲ 数据结构和算法的基本概念

这一部分介绍一些贯穿于整本书的基本概念。

1．1 针对考研数据结构的代码书写规范以及 C&C++语言基础

1．1．1 考研综合应用题中算法设计部分的代码书写规范

要在答题纸上快速的写出能让阅卷老师满意的代码，是有些技巧的，这与写出能在编译器上编译通过的代码有所不同。为了说明这一点我们首先看一个例题：

设将 n（n>1）个整数存放到一维数组 R 中。设计一个算法，将 R 中的序列循环左移 P（0<P<n）个位置，即将 R 中的数据由 $\{X_0,X_1.....X_{n-1}\}$ 变换为

{X_p,X_{p-1},.....,X_{n-1},X_0,X_1...X_{p-1}}。要求：写出本题的算法描述。

分析：

本题不难，要实现 R 中序列循环左移 P 个位置，只需先将 R 中前 P 个元素逆置，再将剩下的元素逆置，最后将 R 中所有的元素再整体做一次逆置操作即可。本题算法描述如下：

```
#include<iostream>                                          //1
#define N 50                                               //2
using namespace std;                                       //3
void Reverse(int R[],int l,int r)                          //4
{                                                          //5
   int i,j;                                                //6
   int temp;                                               //7
   for(i=l,j=r;i<j;i++,j--)                                //8
  {                                                        //9
     temp=R[i];                                           //10
     R[i]=R[j];                                           //11
     R[j]=temp;                                           //12
   }                                                       //13
}                                                          //14
 void RCR(int R[],int n,int p)                             //15
 {                                                         //16
    if(p<=0||p>=n)                                         //17
      cout<<"ERROR"<<endl;                                 //18
    else                                                   //19
    {                                                      //20
       Reverse(R,0,p-1);                                   //21
       Reverse(R,p,n-1);                                   //22
       Reverse(R,0,n-1);                                   //23
    }                                                      //24
 }                                                         //25
   int main()                                              //26
   {                                                       //27
     int L,i;                                              //28
     int R[N],n;                                           //29
     cin>>L;                                               //30
     cin>>n;                                               //31
     for(i=0;i<=n-1;i++)                                   //32
       cin>>R[i];                                          //33
    RCR(R,n,L);                                            //34
    for(i=0;i<=n-1;i++)                                    //35
      cout<<R[i]<<" ";                                     //36
      cout<<endl;                                          //37
     return 0;                                             //38
 }                                                         //39
```

以上程序段，是一段完整的可以在编译器下编译运行的程序，程序比较长，对于考试答卷，完全没有必要这么写。

第 1 和 3 句，在我们大学里写的程序中，几乎都要用到，是耳熟能详的，研究生考试这种选拔考试不会用这种东西来区分学生的优劣，因此答题过程中没必要写，去掉。

第 2 句，定义了一个常量，如果你的题目中要用一个常量，在你用到的地方加上一句注释，说明某某常量之前已经定义即可。而没必要再跑到前边去补上一句#define XX XX，因为试卷的答题纸，不是编译器，插入语句不是那么方便，为了考试的时候节省时间且使得试卷整洁，这是最好的解决办法，因此第 2 句去掉。

第 26 到第 39 是主函数部分，你之前声明的函数（第 4 到第 25 句）在这里调用。在答题中，我们只需要写出自己的函数说明（4 到 25 句），写清楚函数的接口（何为接口，下

边会细致讲解）即可，答卷老师就知道你已经做好了可以解决这个题目的工具（即函数）并且说明了工具的使用方法（即函数接口），这样别人就会用这个工具来解决问题，而没必要你把它用给别人看（主函数中的调用，就是用这个函数解决问题的过程），因此第 26 到第 39 句可以去掉。

经过以上删减，就是以下程序段了，看着是不是简洁了很多？

```
void Reverse(int R[],int l,int r)                                    //1
{                                                                    //2
   int i,j;                                                          //3
   int temp;                                                         //4
   for(i=l,j=r;i<j;i++,j--)                                          //5
   {                                                                 //6
      temp=R[i];                                                     //7
      R[i]=R[j];                                                     //8
      R[j]=temp;                                                     //9
    }                                                                //10
 }                                                                   //11
 void RCR(int R[],int n,int p)                                       //12
 {                                                                   //13
      if(p<=0||p>=n)                                                 //14
       cout<<"ERROR"<<endl;                                          //15
      else                                                           //16
      {                                                              //17
         Reverse(R,0,p-1);                                           //18
         Reverse(R,p,n-1);                                           //19
         Reverse(R,0,n-1);                                           //20
       }                                                             //21
  }                                                                  //22
```

这里来说一下函数的接口。假如上述函数是一台机器，可以用原材料来加工成成品。那么接口就可以理解成原材料的入口，或成品的出口。比如上述成序语段的第 12 句：RCR(int R[],int n,int p)就包含一个接口，它是原材料的一个入口。括号里所描述的就是原材料类型以及名称，是将来函数被调用的时候所要放进去的东西；是在告诉别人，我要三个原材料，第一个是个 int 型的数组；第二个是一个 int 型的变量；第三个也是一个 int 型的变量。第 15 句，cout<<"ERROR"<<endl;也是一个接口，它会在输出设备上打印出一个 ERROR，用来提示用户这里出错了，这算是成品的出口，这里的成品就是一个提示。同时，第 1 句中传入 int 型数组的地方也可以理解为一个产品的出口，因为从这里传入的数组的内容，将在函数执行完后被加工成我们想要的内容。通过以上说明我们可以把接口理解为**用户和函数打交道的地方，通过接口，用户输入了自己的数据，得到了自己想要的结果。**

到这里我们可以知道考研综合应用中算法设计题中的代码部分重点需要写哪些内容了，即**只需写出一个或多个可以解决问题的有着清楚接口描述的函数**即可。

1．1．2 考研中的 C&C++语言基础杂谈

之所以这个标题是 C&C++语言，而不是单一的一种，是因为本书有些程序的书写包含了两者的语法。对于考试答题来说，C++不因为它是 C 语言的升级版就能取代 C。C 和 C++是各有所长的，我们从两者中挑选出来对考研答卷有利的部分，组合起来应用。下边就来具体介绍针对考研数据结构的 C 和 C++语言基础。

1．数据类型

对于基本的数据类型，例如：整型 int,long,……（考研中涉及到处理整数的题目如果没有特别要求用 int 足够了），字符型 char，浮点型 float，double……（对于处理小数的问题，题目没有特殊要求的请况下用 float 就足够了）。这些大家大都了解，就不再

具体讲解，这里主要讲解的是**结构型**和**指针型**。

（1）结构型：

所谓结构型，说的通俗点就是用户自己制作的数据类型。其实我们常用的数组也是用户自己制作的数据类型，数组是多个相同数据类型的变量组合起来的，比如下边这句

```
int a[MAX];//MAX 是已经定义的常量
```

就定义了一个数组，名字为 a。其实就是连续申请了 MAX 个整型变量摆在一起，其中各整型变量之间的位置关系通过数组下标来反映，这不难理解。如果我想制作一个这样的数组，第一个变量是整形变量，第二个变量是字符型变量，第三个变量是浮点型变量，怎么办呢？这里就用到结构体了，**结构体就是系统提供给程序员制作新的数据类型的一种机制，即可以用系统已经有的不同的基本数据类型或用户定义的结构型，组合成用户需要的复杂数据类型。**

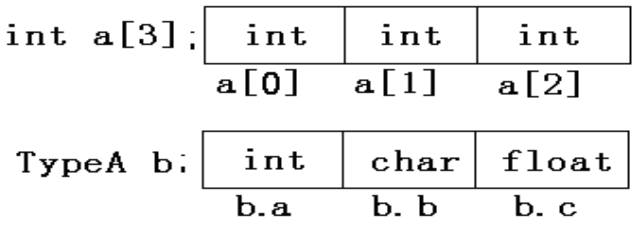
比如上边提到的要制作一个由不同类型的变量组成的数组可以这么构造：

```
typedef struct
{
   int    a;
   char   b;
   float  c;
}TypeA;
```

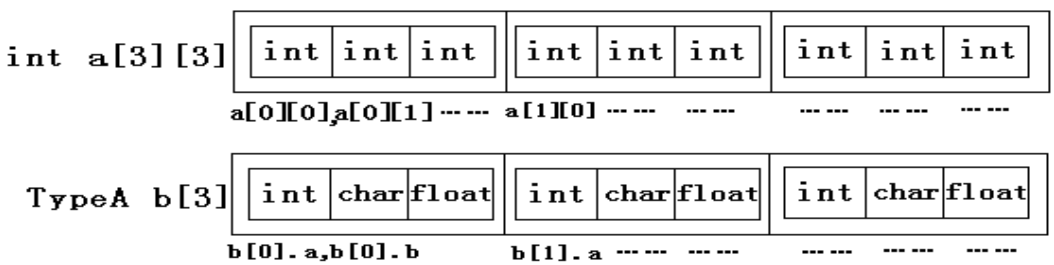
上边的语句制造了一个新的数据类型，TypeA 型。int b[3];申请了一个数组，名字为 b，由 3 个整形的分量组成。而 TypeA a;同样可以认为申请了一个数组，名为 a，只不过组成 a 数组的 3 个分量是不同类型的。对于数组 b，b[0],b[1],b[2]分别代表数组中第一，第二，第三个元素的值。而对于结构体 a，a.a,a.b,a.c 分别对应于结构体变量 a 中第一，第二，第三个元素的值，两者十分相似。

再看这样一个定义 TypeA a[3];这句定义了一个数组,由三个 TypeA 型的元素组成。前边我们知道 TypeA 是结构型，它含有 3 个分量（其实应该叫做结构体的成员，这里为了类比，将它叫做分量），因此 a 数组中的每个元素都是结构型且每个元素都有 3 个分量，它可以类比为哪一种数组呢？显然我们可以把它类比成一个二维数组，比如 int b[3][3];定义了一个名字为 b 的二维数组。二维数组可以看成其数组元素是一维数组的一维数组，如果把 b 看成一个一维数组的话，其中的每个数组元素都有 3 个分量，与 a 数组不同的地方在于，b 中每个元素的 3 个分量是相同类型的，而 a 数组中每个元素的三个分量是不同数据类型的。b 数组取第一个元素的第一个分量的值的写法为 b[0][0]，对应到 a 数组则为 a[0].a。

上边的类比关系可以通过图 1.1 来形象的说明。



结构体和一维数组的对比



数组，和结构体数组的比较

图 1.1 结构体与数组的类比

（2）指针型

指针型和结构型一样，是比较难理解的部分。对于其他类型的变量，变量里所装的东西是数据元素的内容，而指针型变量内部装的是变量的地址，通过它可以找出这个变量在内存中的位置，就像一个指示方向的指针，指出了某个变量的位置，因此叫做指针型。

指针型的定义方法对每种数据类型有特定的写法。有专门指向 int 型变量的指针，有专门指向 char 型变量的指针，等等。对于每种变量，指针的定义方法有相似的规则，如以下语句：

```
int *a;     //对比一下定义 int 型变量的语句： int a;
char *b;    //对比一下定义 char 型变量的语句： char b;
float *c;   //对比一下定义 float 型变量的语句：float c;
TypeA *d;   //对比一下定义 TypeA 型变量的语句：TypeA d;
```

上边四句分别定义了指向整型变量的指针 a，指向字符型变量的指针 b，指向浮点型变量的指针 c，和指向 TypeA 型变量的指针 d。我们看到比起之前所讲其他变量的定义，指针型变量的定义只是在变量名之前多出一个”*”而已。

如果 a 是个指针型变量，且它已经指向一个变量 b，则 a 中存放了变量 b 所在的地址。*a 就是取变量 b 的内容（x=*a;等价于 x=b;），&b 就是取变量 b 的地址，语句 a=&b;就是将变量 b 的地址存于 a 中，即大家常说的指针 a 指向 b。

指针型在考研中用的最多的就是和结构型结合起来构造结点（如链表的结点，二叉树的结点……）。下边我们就来具体讲讲常用结点的构造，这里的“构造”我们就把它理解成先定义一个结点的结构类型，然后用这个定义好的结构型制作一个结点。这样说可能不太严谨，但是不影响理解。

（3）结点的构造

要构造一种结点，必须先定义结点的结构类型。下边描述了链表结点和二叉树结点结构型的定义方法。

1）链表结点的定义

链表的结点有两个域，一个是数据域，用来存放数据，一个是指针域，用来存放下一个结点的位置，如图 1.2 所示。

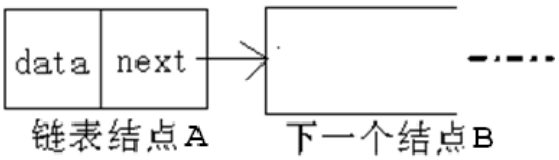


图 1.2 链表节点

因此，链表的结构型定义如下：

```
typedef struct Node
{
    int data;          //这里默认的是 int 型，如需其他类型可直接修改。
    struct Node *next;//指向 Node 型变量的指针
}Node;
```

上边这个结构型的名字为 Node，因为组成此结构体的成员中有一个是指向和自己类型相同的变量的指针，所以内部要用自己来定义这个指针，所以写成 struct Node *next;这里指出，**凡是结构型（假设名为 a）内部有这样的指针型（假设名为 b），即 b 是用来存**

放和 a 类型相同的结构体变量地址的指针型（如图 1.2 中，结点 A 的指针 next，next 所指的结点 B 与结点 A 是属于同一结构型的），则在定义 a 的 typedef struct 语句之后都要加上 a 这个结构型的名字，如上述结构体定义中粗体的 **Node**。与之前定义的结构型 TypeA 进行比较一下，会发现这里的结构型 Node 在定义方法上的不同。

有的书上在把上述链表结点结构定义写成如下形式：

```
typedef struct node
{
    ……
    ……
}Node;
```

我们可以发现，有一个"node"和一个"Node"，即结构体定义中的上下两个名称不同。其实对于考研来说这样写除了增加记忆负担之外，就没别的，所以我希望考生在造结点结构型的时候，上下写成一致的名称。

2）二叉树节点的定义

在链表结点结构型的基础上，再加上一个指向自己同一类型变量的指针域即为二叉树结点结构型，如下：

```
typedef struct BTNode
{
    int   data;
    struct BTNode *lchild;
    struct BTNode *rchild;
}BTNode;
```

在考研数据结构中，只需要熟练掌握以上两种结点的定义方法，其他的结点都是由这两种衍生而来的（其实二叉树结点的定义也是由链表结点的定义衍生而来，二叉树结点只不过比链表结点多了一个指针而已），无需特意的去记忆。

说明：对于结构型，用来实现构造结点的语法有很多不同的表述，我们完全没必要全部掌握。刚刚上边讲到的那些语法用来构造结点已经足够用了，所以我这里建议大家，熟练掌握以上两种构造结点的结构体定义的写法，其他的写法不予理睬。有些语法对我们考试来说复杂又没有意义，比如上边二叉树节点的定义有些书上这样写：

```
typedef struct BTNode
{
    int   data;
    struct BTNode *lchild;
    struct BTNode *rchild;
}BTNode,*btnode;
```

可以看到在最后又多了个*btnode，其实在定义一个结点指针 p 的时候，BTNode *p;等价于 btnode p;对于定义结点指针，BTNode *p;这种写法是何等的顺理成章，因为它继承了之前 int *a;char *b;char *c;TypeA *d;这些指针定义的一般规律，使得我们记忆起来非常方便，何苦再弄个 btnode p;这种写法来增加不必要的记忆负担呢？因此在考研中我们不采取这种方法，对于上边的结构体定义，删去*btnode，统一一个 BTNode，就可以解决所有问题。

除了上边我提到的那种出力不讨好的写法，还有更恶心且没用的写法，这里就不说了，

总之本书的目的就是要帮你从垃圾堆里挑出适合考研的宝贝。

通过以上的讲解，我们知道了链表结点和二叉树结点的定义方法。结构型定义好之后，就要用它来制作新结点了。

以二叉树结点的制作为例，有以下两种写法：

① BTNode BT;

② BTNode *BT;

BT=(BTNode*)malloc(sizeof(BTNode)); **//此句要熟练掌握**

①中只有一句就制作了一个结点，而②中需要两句，比①要繁琐，但是考研中用的最多的是②。②的执行过程是这样：先定义一个结点的指针 BT，然后用 malloc 函数来申请一个结点的内存空间，最后让指针 BT 指向这片内存空间，就完成了一个结点的制作。②中的第二句就是用系统已有的 malloc() 函数申请新结点所需内存空间的方法，考研数据结构中所有类型结点的内存分配都可用 malloc()来完成，模式固定，容易记忆。

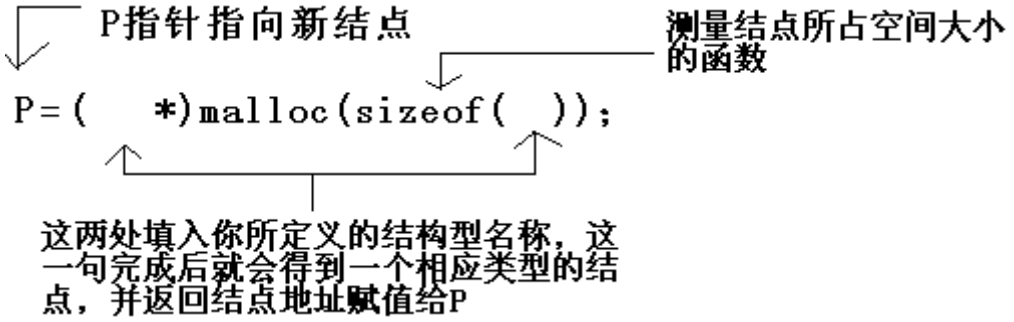


图 1.3 结点空间申请函数

图 1.3 就是申请一个结点空间，并用一个指针（图中为 p）指向这个空间的标准模板。考生需要将这个模板背下来，以后制作一个新结点的时候，只要把结点结构型的名称填入上图括号的空白处即可。这个语法是 C 语言就有的，C++语言还有其他方法，不需要掌握那么多，对于考研来说有这一个足够了。

②句中的 BT 是个指针型变量，用它来存储刚制作好的结点的地址。因 BT 是变量，虽然现在 BT 指向了刚生成的结点，但是在以后必要的时候 BT 可以离开这个结点转而指向其他结点。而①句则不行，①中的 BT 就是某个结点的名字，一旦定义好，它就不能脱离这个结点。从这里就看到②比①更灵活，因此②用的多，并且②完全可以取代①（②中 BT 的值不改变的话就相当于 ①）。

对于①和②中的 BT 取分量的操作也是不同的。对于①如果我想取其 data 域的值付给 x，则应该写成 x=BT.data; 而对于②则应该写成 x=BT->data;一般来说，用结构体变量直接取分量，其操作用”.”，用指向结构体变量的指针来取分量，其操作用”->”。

这里再扩展一点，前边我们提到，如果 p 是指针（假设已经指向 x），*p 就是取这个变量的值，a=*p;等价于 a=x;那么对于②中的 BT 指针，怎么用”.”来取其 data 值呢？类比 p，*BT 就是 BT 指向的变量，因此可以写成(*BT).data;（(*BT).data;与 BT->data 是等价的）。注意 *BT 外边要用括号括起来，不要写成*BT.data。在 C 或 C++语言中这是一种好的习惯，**在你不知道系统默认的运算符优先级的情况下，你最好依照自己所期望的运算顺序加上括号**。有可能这个括号加上是多余的，但是为了减少错误，这种做法是必要的。对于与刚才那句，我所期望的运算顺序是先算*BT，即用”*”先将 BT 变成它所指的变量，然后再用”.”取分量值。因此写成(*BT).data。比如这样一个式子 a*b/c，假设你不知道系统会默认先算乘再算除，而你所期望的运算优先顺序是先算乘再算除，为了减少错误，

你最好是把它写成(a*b)/c，即便这里的括号是多余的。

（4）关于 typedef 和#define

1）typedef

有的书上在定义变量的时候会出现一些你在程序设计教材中从来没见过的诡异的数据类型，比如严奶奶书上就有类似于 Elemtype A;的变量定义语句，这里的 Elemtype 是什么类型，新来的同学常常会一头雾水。要说明这个问题，我们先来说明一下 typedef 的用法。一句话，**typedef 就是用来给现有的数据类型起一个新名字的**，我们在结构类型定义时用到过，如 typedef struct{……} TypeA;即为给 “struct{……}”起了一个名字 TypeA，就好比你制作了计算机中的整型，给他起了个名字为 int。并且如果我想给 int 型起个新名字 A,就可以这样写 typedef int A;这样的话定义一个整形变量 x 的时候 A x;就等价于 int x;在考研中 typedef 用的最多的地方就在结构型的定义过程中，其他的地方几乎不用。你可以这样理解 typedef 是用来起名字的，新定义的结构型没有名字，因此用 typedef 给它起个名字是有必要的，但是对于已有的数据类型，如 int，float 等已经有了简洁的名字，还有必要给它起个新名字吗？有必要，但不是在考研数据结构中。为什么有必要，有兴趣的同学可以去查下资料，查完你会发现，typedef 对程序设计的贡献很大，但是对于考研答卷，用处不大，所以大家对其用法不必深究。说到这里大家就明白了，严奶奶的书上之所以有那么多大家不认识的数据类型，只不过是严奶奶悄悄的给我们认识的数据类型起了新名字而已。

2）#define

在严奶奶的书上除了我们没见过的数据类型以外，还有一些东西我们也没见过，比如在一个函数中她会写到 return ERROR; return OK; 之类的语句，对于经常在编译器上写代码的同学，乍一看到这种语句会十分的不爽，立马就会想，ERROR 和 OK 这样的东西能编译通过？或者就是怀疑自己语言水平太差,严奶奶写的程序里边有太多自己看不懂的地方了，信心大减。其实不然，和 typedef 一样，严奶奶悄悄的用#define 语句处理过 ERROR 或者 OK 之类的词了。其实 ERROR 和 OK 就是两个常量，作为函数的返回值，来提示用户函数操作结果的。严奶奶初衷是想把 0,1 这种常作为函数返回标记的数字定义成 ERROR 和 OK（一般出错返回 0，成功返回 1），这样比起数字来更人性化，更容易理解，但结果却适得其反，让新手们更困惑了。#define 对于考研数据结构可以说没有什么贡献，我们只要认得它就行，写程序时一般用不到。比如#define MAX 50 这句，即定义了常量 MAX（此时 x=50;等价于 x=MAX;)。在写程序大题的时候如果你要定义一个数组,如 int A[MAX];加上一句注释：“/*MAX 为已经定义的常量，其值为 50*/”即可。没必要跑到你的程序最前边去加上#define MAX 50 这一句，原因前边已经讲过。

严奶奶的书中有很多用自己加工过的代码书写的程序,和编译器上我们习惯的写法有很大出入，所以对于新手较难理解。本书的作用在很大程度上就是做了一个翻译的角色，不过是站在学生的角度把课本上用过于严谨以及专业化的词语描述的思想用通俗易懂的语言表达给你而已。

2．函数

说明:只要是算法设计题，必用到函数，所以其中的一些注意事项这里有必要说一下。

（1）用函数来缩短代码

如果有一段较长的操作需要在一个函数中反复多次使用,那么你最好把这个操作做成一个函数，在你要用的地方调用它，会节省很多答题空间。比如：

```
void f()
{
   ……//1
```

```
    ……//2
    ……//3
    ……//4
    ……//5
    ……//6
    ……//7
    ……//8
  }
```

这个函数中的 8 句组成了一个操作。这个操作在另一个函数（函数名为 F）中要多次用到。此时我们就可以把这 8 句做成一个函数，当用到的时候调用即可，比如：

```
void F()
{
  f();
    ……
    ……
  f();
    ……
    ……
  f();
}
```

从上边可以看出，如果不用 f()函数，就得把 f()中的 8 行写 3 遍，使得 F()函数很长。

（2）被传入函数的参数是否会改变

```
int a;
void f(int x)
{
  x++;
}
```

上边声明的函数，它需要一个整型变量作为参数，且在自己的函数体中将参数做自增 1 的运算。执行完以下程序段之后 a 的值是多少呢？

```
a=0;   //①
f(a);  //②
```

有些同学可能以为 a 等于 1。这个答案是错误的，可以这样理解，对于函数 f()，在调用他的时候，括号里的变量 a 和句①中的变量 a 并不是同一个变量。在执行句②的时候，变量 a 只是把自己的值赋给了一个在 f()的声明过程中已经定义好的整形变量，可以把这个变量想象为上述声明过程中的 x，即句②的执行过程拆开看来是这样两句：x=a;x++;因此 a 的值在执行完①,②两句之后不变。

如果我想让 a 依照 f()函数体中的操作来改变应该怎么写呢，这里就要用到函数的**引用型**（这种语法是 C++中的，C 中没有，C 中是靠传入变量的地址的方法来实现的，写起来比较麻烦且容易出错，因此这里采用 C++的语法），其函数声明方法如下：

```
void f(int &x)
{
   x++;
}
```

这样就相当于 a 取代了 x 的位置，函数 f()就是在对 a 本身进行操作，执行完①②两句后，a 的值由 0 变为 1。

上边讲到的是对针对普通变量的“引用型”，如果传入的变量是指针型变量，且在函数体内要对传入的指针进行改变，则需写成如下形式：

```
void f(int *&x) //指针型变量在函数体中需要改变的写法。
{
   x++;
}
```

执行完上述函数后，指针 x 的值自增 1。

说明：这种写法很多同学不太熟悉，但是它在树与图的算法中应用广泛，在之后的章节中考生要注意观察其与一般引用型变量的书写差别。

上边是单个变量作为函数参数的情况。如果一个数组作为函数的参数，该怎么写呢？传入的数组是不是也有“引用型”一说呢？对于数组作为函数的参数，这里讲两种情况，一维和二维数组。

一维数组作为参数的函数声明方法：

```
void f(int x[],int n)
{
   ……;
}
```

对于第一个参数位置上的数组的定义只需写出两个中括号即可，不需要限定数组长度（即不需要写成 f(int x[5],int n)，即便是你传入的数组真的是长度为 5），对于第二个参数 n，是写数组作参数的函数的习惯，用来说明将来要传进函数加工的数组元素的个数，并不是指数组的总长度。

二维数组作为参数的函数声明方法：

```
void f(int x[][MAX], int n)
{
  ……;
}
```

如果函数的参数是二维数组，数组的第一个中括号内不需要写上数组长度，而第二个中括号内必须写上数组长度（假设 MAX 是已经定义的常量）。这里需要注意，你所传入的数组第二维长度也得是 MAX，否则出错，比如：

```
void f（int x[][5]）
{
  ……;
}
int a[10][5];
int b[10][3];
f(a);   //参数正确
f(b);   //参数错误
```

要注意的是，将数组作为参数传入函数，函数就是对传入的数组本身进行操作，即如果函数体内涉及到改变数组数据的操作，传入的数组中的数据就会依照函数的操作来改变。因此，对于数组来说，没有“引用型”和“非引用型”之分，可以理解为只要数组作为参数，

都是引用型的。

（3）有返回值的函数

声明一个函数：

```
int f(int a)
{
  return a;
}
```

在这个声明中我们可以看到，有一个 int 在函数名的前边，这个 int 是指函数返回值是 int 型。如果没有返回值，声明函数的时候用 void，前边讲过的函数中已经有所体现。返回值常常用来作为判断函数执行状态（完成还是出错）的标记，或者一个计算的结果。严奶奶的书中出现过类似于下边这样的函数。

```
STATUS f(ELEMTYPE a)
{
   if(a>=0) return ERROR;
   else     return  OK;
}
```

对于一些基础稍差的同学来说，这个函数麻烦了，STATUS,ELEMTYPE,ERROR,OK 这都什么东西，其实严奶奶在离这个函数很远的地方写过这些语句：

```
#define ERROR 1
#define OK 0
typedef STATUS bool      //这句中的 bool 是布尔型，只取两个值
                         //0 和 1，其实用 bool 用 int 型代替就可以，
                         //所以对于考研 bool 用处不大。
typedef ELEMTYPE int
```

在那个函数前边加上这四句是否看懂了呢？看懂后可以把它翻译一下就能写出以下代码：

```
bool f(int a)  //本行可换成 int f(int a)
{
   if(a>=0) return 1;
   else     return  0;
}
```

上边这种写法是不是清楚明白了。严奶奶之所以要将程序写的如此个性，原因有两个，一是上边那个自己另起的类型名或者常量名，都有着实际的意义，STATUS 代表状态，OK 代表程序执行成功，ERROR 代表出错，这样代码写的就更人性化。二是，如果我们在写一个大工程，对于其中的一个变量，在整个工程中都已经用 int 型定义过了，但是工程现在要求修改，将所有 int 型换成 char 型，这下麻烦就大了。如果你写成上边那种形式，将 int 型起个新名字 ELEMTYPE，在整个程序中凡是类似于 int x;的语句都写成 ELEMTYPE x;此时如果要统一更换数据类型，只需将 typedef ELEMTYPE int 这一句中的 int 换成 char 即可，这无疑是十分方便的事情，这就是 typedef 对于程序设计的意义所在（#define 也能达到类似的目的）。但显然的是，这对考研答卷意义不大。

1．2 算法的时间复杂度与空间复杂度分析基础

1．2．1 考研中的算法时间复杂度杂谈

对于这部分，要牢记住一句话：**将算法中基本操作的执行次数作为算法的时间复杂度。**这里我们所讨论的时间复杂度，不是执行完一段程序的总时间，而是其中基本操作的总次数。因此对于一个算法进行时间复杂度分析的要点，无非是明确算法中哪些操作是基本操作，然后计算出基本操作所重复执行的次数。在考试中算法题目里你总能找到一个 n，可以称为问题的规模，比如要处理的数组元素的个数为 n，而基本操作所执行的次数是 n 的一个函数 f(n)（这里的函数是数学中的函数的概念，不是 C 或 C++语言中的函数的概念）。对于求其基本操作执行的次数，就是求函数 f(n)。求出以后我们就可以取出 f(n)中随 n 增大增长最快的项，然后将其系数变为 1 做为时间复杂度的度量，记为 **T(n)=O(f(n)中增长最快的项／此项的系数)**，比如 $f(n)=2n^3+4n^2+100$，则其时间复杂度为为 $T(n)=O(2n^3/2)=O(n^3)$。其实计算算法的时间复杂度，就是给出相应的数量级，当 f(n)与 n 无关时，时间复杂度 T(n)=O(1)；当 f(n)与 n 是线性关系时，T(n)=O(n)；是平方关系时，$T(n)=O(n^2)$；以此类推。

说明：考研中常常要比较各种时间复杂度的大小，常用的比较关系如下：

$$O(1)\leqslant O(\log_2(n))\leqslant O(n)\leqslant n\log_2(n)\leqslant O(n^2)\leqslant O(n^3)\leqslant \ldots\ldots\leqslant O(n^k)\leqslant O(2^n)$$

通过以上分析我们总结出计算一个算法时间复杂度的步骤如下：

（1） 确定算法中的基本操作，以及问题的规模。

（2）根据基本操作执行情况计算出规模 n 的函数 f(n)，并确定时间复杂度为 T(n)=O(f(n)中增长最快的项/此项的系数)。

注意：有的算法中基本操作执行次数跟初始输入的数据有关。如果题目不做特殊要求，一般我们依照使得基本操作执行次数最多的输入来计算时间复杂度，即将最坏的情况作为算法时间复杂度的度量。

1．2．2 例题选讲

例题 1：求出以下算法的时间复杂度。

```
void fun(int n)
{
  int i=1,j=100;
  while(i<n)
  {
    j++;
    i+=2;
  }
}
```

分析：

第一步：找出基本操作，确定规模 n。

①找基本操作（所谓基本操作，即其重复执行次数和算法的执行时间成正比的操作，通俗点说，这种操作组成了算法，当它们都执行完的时候算法也结束了，**多数情况下我们取最深层循环内的语句所描述的操作作为基本操作**），显然题目中 j++;与 i+=2;这两行都可以

作为基本操作。

②确定规模，由循环条件 i<n 可以知道，循环执行的次数，即基本操作执行的次数和参数 n 有关，因此参数 n 就是我们所说的规模 n。

第二步：计算出 n 的函数 f(n)。

显然，n 确定以后，循环的结束与否与 i 有关，i 的初值为 1，每次自增 2，假设 i 自增 m 次后循环结束，则 i 最后的值为 1+2×m，因此有 1+2×m+K=n（其中 K 为一个常数，因为在循环结束时 i 的值稍大于 n,为了方便表述和进一步计算,用 K 将 1+2×m 修正成 n。因为 K 为常数，所以这样做不会影响最终时间复杂度的计算），解得 m=(n-1-K)/2，即 f(n)=(n-1-K)/2，可以发现其中增长最快的项为 n/2,因此时间复杂度 T(n)=O(n)。

例题 2：分析以下算法的时间复杂度。

```
void fun(int n)
{
    int i, j, x=0;
    for(i=1;i<n;i++)
       for(j=i+1;j<=n;j++)
          x++;
 }
```

分析：

x++;处于最内层循环，因此取 x++;做为基本操作。显然 n 为规模。可以算出 x++;的执行次数为 f(n)=n(n-1)/2，变化最快的项为 n^2，因此时间复杂度为 $T(n)=O(n^2)$。

例题 3：分析以下算法的时间复杂度。

```
void fun(int n)
{
    int i=0;s=0;
    while(s<n)
    {
       i++;
       s=s+i;
    }
}
```

分析：

显然 n 为规模,基本操作为 i++;s=s+i;i 与 s 都从 0 开始,假设循环执行 m 次结束，则有 $s_1=1, s_2=1+2=3, s_3=1+2+3=6, \cdots\cdots, s_m=m(m+1)/2$（其中 s_m 为执行到第 m 次的时候 s 的值），则有 m(m+1)/2+K=n，（K 为起修正作用的常数）由求根公式得：

$$m=\frac{-1+\sqrt{8n+1}-8}{2}$$

即：

$$f(n)=\frac{-1+\sqrt{8n+1}-8}{2}$$

由此可知时间复杂度为：

$$T(n)=O(\sqrt{n})$$

说明：在计算时间复杂度的时候有可能会出现这种情况，即对于相同的规模，因输入序列不同会出现不同的时间复杂度，这时我们一般取最坏的情况下的输入序列（即使得基本操作执行次数最多的序列）来计算时间复杂度。

1．2．3 考研中的算法空间复杂度分析

算法的空间复杂度指算法在运行时所需**存储空间的度量**，主要考虑在算法运行过程中**临时占用**的存储空间的大小（和时间复杂度一样，以数量级的形式给出）。

说明：这一部分在理解了各种数据的存储结构及其操作之后更容易理解。因此对于这一部分，将在后边的章节中以题目的形式给出讲解。

1．3 数据结构和算法的基本概念

1．3．1 数据结构的基本概念（不需要刻意的去记忆这些内容，联系生活实际去理解即可。在以后的学习过程中，如果碰到不熟悉的概念，来这里查一查就可以了。）

1．数据

数据是对客观事物的符号表示，在计算机科学中是指所有能输入到计算机中并且被计算机程序处理的符号的总称。例如：整数、实数、和字符串都是数据。

2．数据元素

数据元素是数据的基本单位，在计算机程序中通常将其作为一个整体进行考虑和处理。有时，一个数据元素可由若干个数据项组成；例如，一本书的书目信息为一个数据元素，而书目信息的每一项（如书名，作者名等）为一个数据项。

3．数据项

数据项是数据结构中讨论的最小单位，是数据记录中最基本的，不可分的数据单位。

4．数据对象

数据对象是性质相同的数据元素的集合，是数据的一个子集。例如，大写字母就是一个数据对象，大写字母数据对象是集合{'A','B'……'Z'}。

5．数据结构

数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。数据结构包括 3 方面的内容：逻辑结构，存储结构和对数据的运算。

6．数据的逻辑结构

数据的逻辑结构是对数据之间关系的描述，它与数据的存储结构无关，同一种逻辑结构可以有多种存储结构。归纳起来数据的逻辑结构主要有两大类。

（1） 线性结构

简单地说，线性结构是一个数据元素的有序（次序）集合。它有四个基本特征：

1）集合中必存在唯一的一个"第一个元素"。

2）集合中必存在唯一的一个"最后的元素"。

3）除最后元素之外，其它数据元素均有唯一的"后继"。

4）除第一元素之外，其它数据元素均有唯一的"前驱"。

数据结构中线性结构指的是数据元素之间存在着"一对一"的线性关系的数据结构。

如（a1,a2,a3,.....,an），a1 为第一个元素，an 为最后一个元素，此集合即为一个线性结构的集合。

（2） 非线性结构

与线性结构不同，非线性结构中的结点存在着一对多的关系，它又可以细分为树形结构和图形结构。

7．数据的物理结构

数据的物理结构又称为存储结构，是数据的逻辑结构在计算机中的表示（又称映像）。它包括数据元素的表示和关系的表示。当数据元素是由若干数据项构成的时候，数据项的表示称为数据域；比如一个链表结点，结点包含值域和指针域，这里结点可以看做一个数据元素，其中的值域和指针域都是这个数据元素的数据域。

数据元素之间的关系在计算机中有两种不同的表示方法：顺序映像和非顺序映像。对应的两种不同的存储结构分别是顺序存储结构和链式存储结构。顺序映像是借助数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系；非顺序映像是借助指针表示数据元素之间的逻辑关系。实际上，在数据结构中有以下 4 种常用的存储方法。

（1）顺序存储方法

顺序存储结构是存储结构类型中的一种，该结构是把逻辑上相邻的结点存储在物理位置上相邻的存储单元中，结点之间的逻辑关系由存储单元的邻接关系来体现。由此得到的存储结构为顺序存储结构，通常顺序存储结构式借助于计算机程序设计语言（例如 C/C++）的数组来描述的。

（2）链式存储方法

该方法不要求逻辑上相邻的结点在物理位置上亦相邻，结点间的逻辑关系是由附加的指针字段表示的。由此得到的存储表示称为链式存储结构，通常借助于计算机程序设计语言（例如 C/C++）的指针类型来描述它。

（3）索引存储方法

该方法在存储结点信息时除建立存储结点信息外，还建立附加的索引表来标识结点的地址。索引项的一般形式一般是<关键字，地址>。关键字标识唯一一个结点：地址作为指向结点的指针。

（4）散列（或哈希）存储方法

该方法的基本思想是根据结点的关键字通过哈希函数直接计算出该结点的存储地址。这种存储方法本质上是顺序存储方法的扩展。

8．数据类型和变量

数据类型是一个值的集合以及定义在这个值集上的一组操作。

变量是用来存储值的所在处，它们有名字和数据类型。变量的数据类型决定了如何将代表这些值的位存储到计算机的内存中。在声明变量时也可指定它的数据类型。所有变量都具有数据类型，以决定能够存储哪种数据。

1．3．2 算法的基本概念

1．算法

算法可以理解为有基本运算及规定的运算顺序所构成的完整的解题步骤。或者看成按照要求设计好的有限的确切的计算序列。

2．算法的特性

一个算法应该具有以下五个重要的特征：

（1）有穷性

一个算法必须保证执行有限步之后结束。

（2）确定性

算法的每一步骤必须有确定的定义。

（3）输入

一个算法有 0 个或多个输入，以刻画运算对象的初始情况，所谓 0 个输入是指算法本身确定了初始条件。

（4）输出

一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的。

（5）可行性

算法中的所有操作都必须可以通过已经实现的基本操作机型运算，并在有限次内实现，而且人们用笔和纸做有限次运算后也可完成。

3．算法的设计目标

算法设计目标为正确性、可读性、健壮性和算法效率。其中算法效率通过算法的时间复杂度和空间复杂度来描述。

（1）正确性

要求算法能够正确地执行预先规定的功能和性能要求。这是最重要也是最基本的标准。

（2）可读性

要求算法易于人的理解。

（3）健壮性

要求算法有很好的容错性，能够对不合理的数据进行检查。

（4）高效率与低存储量需求

算法的效率主要是指算法的执行时间。对于同一个问题如果有多种算法可以求解，执行时间短的算法效率高。算法的存储量指的是算法执行过程中所需要的最大存储空间。高效率和低存储量这两者都与问题的规模有关。

习题心选

一、选择题

1．算法的计算量的大小称为算法的（　　）。

A．效率　　B．复杂性　　C．现实性　　D．难度

2．算法的时间复杂度取决于（　　）

A．问题的规模　　B．待处理数据的初态　　C．A 和 B

3.计算机算法指的是（1），它必须具备（2） 这三个特性。

(1)A．计算方法　　B．排序方法　　C．解决问题的步骤序列　　D．调度方法

(2)A．可执行性、可移植性、可扩充性　　B．可执行性、确定性、有穷性

C．确定性、有穷性、稳定性　　D．易读性、稳定性、安全性

4．一个算法应该是（　　）。

A．程序　　B．问题求解步骤的描述　　C．要满足五个基本特性　　D．A 和 C

5．下面关于算法说法错误的是（　　）。

A．算法最终必须由计算机程序实现

B．为解决某问题的算法同为该问题编写的程序含义是相同的

C．算法的可行性是指指令不能有二义性

D．以上几个都是错误的

6．下面说法错误的是（　　）。

（1）算法原地工作的含义是指不需要任何额外的辅助空间

（2）在相同的规模 n 下，复杂度 O(n)的算法在时间上总是优于复杂度 $O(2^n)$的算法
（3）所谓时间复杂度是指最坏情况下，估算算法执行时间的一个上界
（4）同一个算法，实现语言的级别越高，执行效率就越低

A. (1) B.(1),(2) C.(1),(4) D.(3)

7. 从逻辑上可以把数据结构分为（ ）两大类。
A. 动态结构、静态结构 B. 顺序结构、链式结构
C. 线性结构、非线性结构 D. 初等结构、构造型结构

8. 以下哪一个术语与数据的存储结构无关？（ ）。
A. 栈 B.哈希表 C.线索树 D.双向链表 E.循环队列

9. 在下面的程序段中，对 x 的赋值语句的频度为（ ）。

```
for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
      x++;
```

A. O(2n) B. O(n) C. $O(n^2)$ D. $O(\log_2^n)$

10. 程序段
```
for(i=n-1;i>=1;i--)
      for(j=1;j<=i;j++)
           if(A[j]>A[j+1]) A[j]与A[j+1]对换;
```
其中 n 为正整数，则最后一行的语句频度在最坏情况下是（ ）。
A. O（n） B. O(nlogn) C. $O(n^3)$ D. $O(n^2)$

11. 以下数据结构中，（ ）是非线性数据结构
A. 树 B. 队 C. 栈

12. 连续存储设计时，存储单元的地址（ ）。
A. 一定连续 B. 一定不连续 C. 不一定连续 D. 部分连续，部分不连续

13. 以下属于逻辑结构的是（ ）。
A. 顺序表 B. 哈希表 C.有序表 D. 单链表

二. 综合应用题

1. 有下列运行时间函数：
（1）$f_1(n)=1000$; （2）$f_2(n)=n^2+1000n$; （3）$f_3(n)=3n^3+100n^2+n+1$;
分别写出相应的大 O 表示的运算时间。

2. 下面函数 mergesort 执行的时间复杂度为多少？假设函数调用被写为 mergesort(1,n),merge 函数时间复杂度为 O(n)。

```
void mergesort(int i,int j)
{
    int m;
    if(i!=j)
    {
        mergesort(i,m);
        mergesort(m+1,j);
        merge(i,j,m);//本函数时间复杂度为O(n)
    }
}
```

习题心讲

选择题：

1.B

本题考查算法时间复杂度的定义。算法中基本操作的重复执行次数就是算法的计算量，将其大小作为算法的时间复杂度，因此选 B。

2.C

本题考查算法时间复杂度的定义。算法时间复杂度即为基本操作执行次数，显然问题规模越大，基本操作的次数越多，因此时间复杂度与规模有关。在相同的规模下，与数据初态也有关，比如两个数相乘，有一个因子为 0 时的计算速度显然要比两个因子都非 0 的情况要快。本题选 C。

3. C、B

本章算法基本概念中已讲过。

4.B

本章算法基本概念中已讲过。

5.D

本题考查算法的概念。

选项 A　计算机程序只是实现算法的一种手段，人用手工也可以完成。

选项 B　算法可以理解为由基本运算及规定的运算顺序所构成的完整的解题步骤。程序是为实现特定目标或解决特定问题而用计算机语言编写的命令序列的集合。两者显然是不同的概念。

选项 C　明显错误，课本概念中已经讲过。

6.C

本题考查算法的时间复杂度和空间复杂度的相关知识。

（1）一个可执行程序除了需要内存空间来寄存本身的指令、常数、变量和输入数据外，还需要额外空间，如果这个额外空间相对于问题的规模（输入数据）来说是个常数，那我么们就称之为原地工作。因此（1）不对。

（2）1.2.1 节考研中的算法时间复杂度杂谈中已经讲过。

（3）对于 O(1),O($\log_2$n),O(n)等，O 的形式定义为：若 f(n)是正数 n 的一个函数，则 O(f(n))表示存在一个正常数 M，使得当 n≥n_0时满足|O(f(n))|小于等于 M×|f(n)|，也就是 O(f(n))给出了函数 f(n)的一个上界。

（4）这句话是严版数据结构上的原句，大多数情况下应该是这样，但是不能说的这么绝对，得看编译链接后的最终的机器指令，这些指令操作的次数越少，说明该语言在某种编译链接环境下效率较高，实际上即使同一种语言在不同的编译环境下，也有可能不同。

综上，本题答案为 C。

7.C

1.3. 1 节数据结构的概念中已经讲过。

8.A

本题考查基本数据结构。

A 项，栈是逻辑结构。

从 1.1.3 节第 7 个讲解中可以知道。

B 项，线索树是在链式存储结构的基础上对树进行线索，与链式存储结构有关。

C 项，双向链表也是说明线性表是以链式结构存储。

D 项，哈希是算法，哈希存储方法本质上是顺序存储方法的扩展。哈希表本质上是顺序

表的扩展。

E 项，循环队列是建立在顺序存储结构上的。

说明：这种题目还有一种比较直观的解法，要判断是否与数据的存储结构无关，只需看看这种结构到底有没有具体到使用顺序存储还是链式存储，如果已经具体到了那就一定是和数据的存储结构有关，比如 A 选项中的栈并没有说明是用顺序栈还是用链栈来实现，所以是逻辑结构。B 选项中的线索树很明显是要用链式来实现（现在不清楚没关系，等学完树那一章就理解了），故与数据的存储结构有关，以此类推。

9.C

本题考查算法时间复杂度的计算。$f(n)=n^2$，因此时间复杂度是 $O(n^2)$。

10.D

本题考查算法时间复杂度的计算。此算法为冒泡排序算法的核心语句，最坏情况下时间复杂度为 $O(n^2)$。

11.A

本题考查基本数据结构。树是一种分支结构，显然不属于线性结构。

12.A

本题考查数据的物理结构。顺序存储结构要开辟一片连续的存储空间，结合 1.1.3 第 7 个讲解可知本题选 A

13.C

本题考查数据的物理结构。有序表指出了表中数据是有一定逻辑顺序排列的，是一种逻辑结构。

综合应用题：

1．答案：

根据 1.1.2 节中公式得：

（1）$T_1(n)=O(1000/1000)=O(1)$

（2）$T_2(n)=O(n^2/1)=O(n^2)$

（3）$T_3(n)=O(3n^3/3)=O(n^3)$

2．分析：

显然规模为 n，基本操在 merge 函数中，merge 时间复杂度为 O(n)因此 merge 内基本操作次数可设为 cn，mergesort 函数基本操作次数设为 f(n)，则有：

$f(n)=2f(n/2)+cn$

$=2^2f(n/4)+2cn$

$=2^3f(n/8)+3cn$

$=\cdots\cdots$

$=2^kf(n/(2^k))+kcn$……①

由 mergesort 函数可知，$f(1)=O(1)$……②

由①②可知，当 $n=2^k$ 即 $k=\log_2 n$ 时

$f(n)=n+kcn\log_2 n$

因此时间复杂度 $T(n)=O(n\log_2 n)$

作者的话：假如你是一个很新的新手（至少要稍微有一点 C 或 C++语言基础，如果连这个都没有的话，先去读谭浩强的 C），到这里为止，我已经把考研数据结构要用到的全部基本功教给你了，对于考研数据结构，要想拿高分，有很多学习的风格，本书则走了一种让广大考生都容易接受的风格，希望读者能适应这种风格，这样你的学习会变的轻松。下面就让我们从下一章开始，把考纲所要求的知识点一一击破。

第二章 线性表

大纲要求

▲ 线性表的定义和基本操作

▲ 线性表的实现

1．顺序存储结构
2．链式存储结构
3．线性表的应用(这一部分通过线性表算法中的各种题目来讲解，不单独作为一节)

说明：本章大纲要求过于简略，有些知识点大纲没有明文写出，但是掌握了这些知识点对本章以及其他章中很多知识点的理解都有帮助。因此本章会添加一些必须的知识点，以帮助你更好的理解，虽然大纲中没有涉及这些知识点。

2．1 线性表的基本概念与实现

1.线性表的定义

线性表是具有**相同特性**数据元素的一个**有限**序列。该序列中所含元素的**个数**叫做线性表的长度，用 n 表示（n≥0）；注意 n 可以等于零，表示线性表是一个空表，空表也可以作为一个线性表。

线性表是一种简单的数据结构，我们可以把它想象成一队学生。学生人数对应了线性表的长度，学生人数是有限的，这里体现了线性表是一个有限序列；队中所有人的身份都是学生,这里体现了线性表中的数据元素具有相同的特性;线性表可以是有序的也可以是无序的，如果学生是按照身高来排队，矮在前，高在后，这就体现了线性表的有序性。

2.线性表的逻辑特性

继续拿定义中的例子来说明。在一队学生中，只有一个学生在队头，同样只有一个学生在队尾。在队头的学生的前面没有其他学生，在队尾的学生的后边也没有其他学生。除了队头和队尾的学生以外，对于其他的每一个学生，紧挨着站在其前后的学生都只有一个，这是很显然的事情。线性表也是这样，只有一个表头元素，只有一个表尾元素，表头元素没有前驱，表尾元素没有后继，除表头和表尾元素之外其他元素只有一个直接前驱，也只有一个直接后继。以上就是线性表的逻辑特性。

3.线性表的存储结构

线性表的存储结构有**顺序存储结构**和**链式存储结构**两种。前者称为**顺序表**，后者称为**链表**。下边就通过对比来介绍这两种存储结构。

（1）顺序表

顺序表就是把线性表中的所有元素按照其逻辑顺序，依次存储到存储器中从指定存储位置开始的一块**连续**的存储空间中。这样线性表中第一个元素的存储位置就是指定的存储位置，第 i+1 个元素的存储位置紧接在第 i 个元素的存储位置的后面。

（2）链表

在链表存储中，每个结点不仅包含所存元素本身的信息，还包含元素之间逻辑关系的信息，即前驱结点包含后继结点的地址信息，这样就可以通过前驱结点中的地址信息方便的找到后继结点的位置。

两种存储结构的比较：

顺序表就好像图 2.1(a)所示的一排房间，每个房间左边的数字就是该房间离 0 点的距离同时也代表了房间号，房间的长度为 1。因此我们只要知道 0 点的位置，然后通过房间号就马上可以找到任何一个房间的位置，这就是顺序表的第一个特性，**随机访问特性**。由下图我们还可以看出，5 个房间所占用的地皮是紧挨着的，即连续的占用了一片空间，并且地皮的块数 6 是确定的，当我们在地皮上布置新的房间或者拆掉老的房间（即对顺序表的操作过程中）地皮的块数不会增加也不会减少。这就是顺序表的第二个特性，即顺序表要求**占用连续的存储空间**，存储分配只能预先进行，即**静态分配**，一旦分配好了，在对其操作过程中不变。

再看链表，如图所示，房间是散落存在的，每个房间的右边有走向下一个房间的方向指示箭头。因此如果我们想访问最后一个房间，就必须从第一个房间开始，依次走过前三个房间才能来到最后一个房间，而不能直接得出最后一个房间的位置，即链表**不支持随机访问**。通过图 2.1(b)我们还可以知道，链表中每一个结点需要划出一部分空间来存储指向下一个结点位置的指针，因此链表中**结点的存储空间利用率较之顺序表稍低一些**。链表中结点是散落的分布在存储器中的，所以链表支持存储空间的**动态分配**，即在需要新的结点时再进行空间划分，而不需要一次性的划分所有所需空间给链表。

图 2.1(a)顺序表中最右边的一个表结点空间代表没有被利用(即顺序表还有剩余空间来注入新数据)，如果我们想在 1 号房间和 2 号房间之间插入一个房间，则必须将 2 号以后的房间都往后移动一个位置（假设房间是可以随意搬动的），即**顺序表做插入操作的时候要移动多个元素**。而链表就无需这样，如图 2.1(b)的链表，如果想在第一个和第二个房间之间插入一个新房间，则只需改动房间后边的方向指示箭头即可，将第一个房间的箭头指向新插入的房间，然后将新插入的房间的箭头指向第二个房间即可，即**在链表中进行插入操作无需移动元素**。

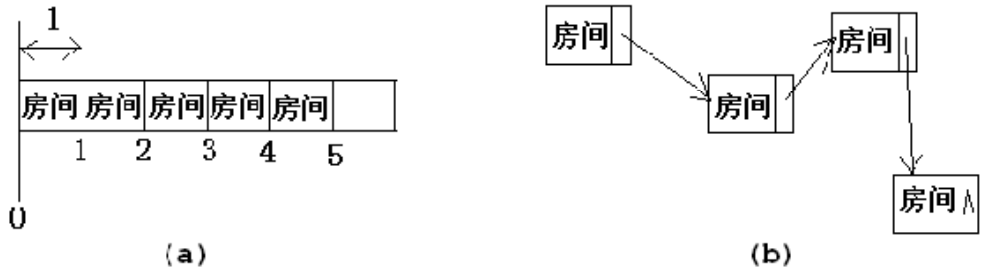


图 2.1 顺序表和链表的比较

链表有如下 5 种形式（在程序题目中用到的链表结点的 c 语言描述将在以后的章节中介绍）：

（1）单链表

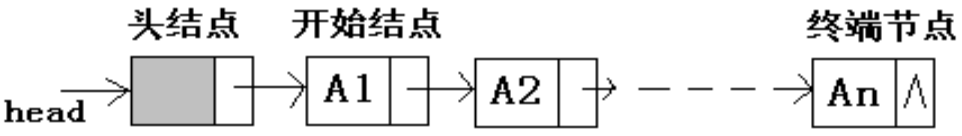


图 2.2 带头结点的单链表

在每个结点中除了包含数据域外，还包含一个指针域，用以指向其后继结点。如图 2.2 所示，即为带头结点的单链表。这里要区分一下带头结点的单链表和不带头结点的单链表。

1）带头结点的单链表中头指针 head 指向头结点，头结点的值域不含任何信息，从头结点的后继结点开始存储信息。头指针 head 始终不等于 NULL，head->next 等于 NULL 的时候链表为空。

2)不带头结点的单链表其中的头指针head直接指向开始结点,即图2.2中的结点A1，当 head 等于 NULL 的时侯链表为空。

总之,两者最明显的区别是,带头结点的单链表有一个结点不存储信息,只是作为标记，而不带头结点的单链表所有结点都存储信息。

注意：在题目中要区分头结点和头指针，不论是带头结点的链表还是不带头结点的链表，头指针都指向链表中第一个结点，即图 2.2 中的 head 指针。而头结点是带头结点的链表中的第一个结点，只作为链表存在的标志，结点内不存信息。

（2）双链表：

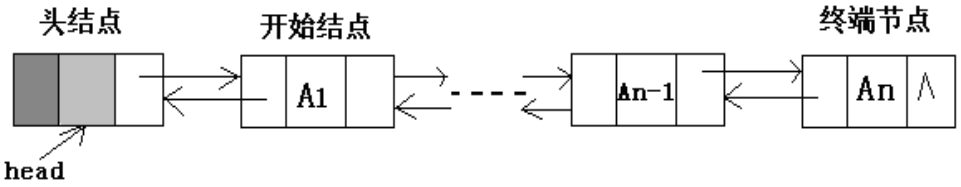


图 2.3 双链表

单链表只能由开始结点走到终端结点，而不能由终端结点反向走到开始结点。如果要求输出从终端结点到开始结点的数据序列，则对于单链表来说操作就非常麻烦。为了解决这类问题我们构造了双链表。如图 2.3 所示，即为带头结点的双链表。双链表就是在单链表结点上增添了一个指针域，指向当前结点的前驱。这样就可以方便的由其后继来找到其前驱，而实现输出终端结点到开始结点的数据序列。

同样，双链表也分为带头结点的双链表和不带头结点的双链表，情况类似于单链表。带头结点的双链表 head->next 为 NULL 的时候链表为空。不带头结点的双链表 head 为 NULL 的时候链表为空。

（3）循环单链表

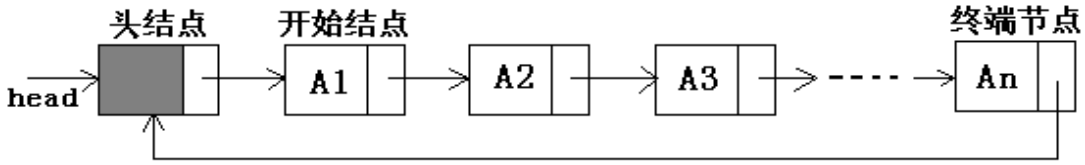


图 2.4 循环链表

知道了单链表的结构之后，循环单链表就显得比较简单了，只要将单链表的最后一个指针域（空指针）指向链表中第一个结点即可（这里之所以说第一个结点而不说是头结点是因为，如果循环单链表是带头结点的则最后一个结点的指针域要指向头结点；如果循环单链表不带头结点，则最后一个指针域要指向开始结点）。如图 2.4 所示，即为带头结点的循环单链表。循环单链表可以实现从任一个结点出发访问链表中任何结点，而单链表从任一结点出发后只能访问这个结点本身及其后边的所有结点。带头结点的循环单链表当 head 等于 head->next;时链表为空；不带头结点的循环单链表当 head 等于 NULL 时链表为空。

（4）循环双链表

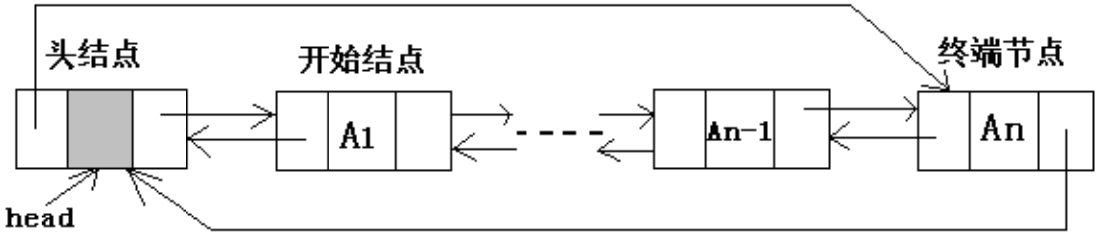
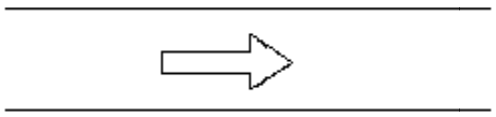


图 2.5 双循环链表

和循环单链表类似，循环双链表的构造源自双链表，即将终端结点的 next 指针指向链表中第一个结点，将链表中第一个结点的 prior 指针指向终端结点，如图 2.5 所示。循环双链表同样有带头结点和不带头结点之分。带头结点的循环双链表当 head->next 和 head->prior 两个指针都等于 head 时链表为空，不带头结点的循环双链表当 head 等于 NULL 的时候为空。

上述前 4 种链表可以用 4 种道路来形象的比喻一下，如以下图组：

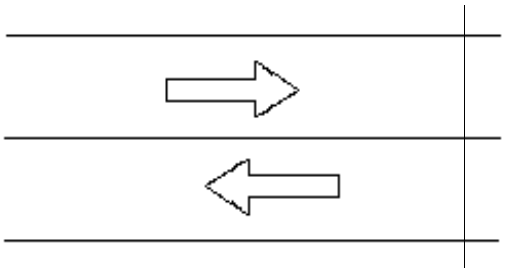
单链表：



(a)

单链表就像图(a)中的单行车道，只允许车辆往一个方向行驶。

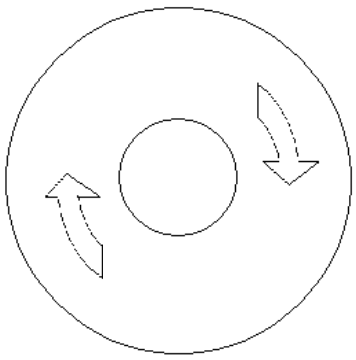
双链表：



(b)

双链表就像图(b)中的双向车道，车辆既可以从左往右行驶，又可以从右向左行驶。

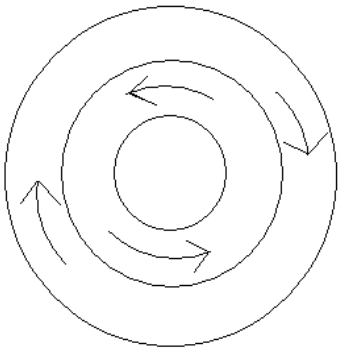
循环单链表：



(c)

循环单链表就像图(c)的环形车道，车辆可沿着一个方向行驶在这条车道上。

循环双链表：



(d)

循环双链表就像图(d)的双向环形车道，车辆可以沿着 2 个方向行驶在这条车道上。

（5）静态链表

这种链表借助一维数组来表示，如图 2.6 所示。

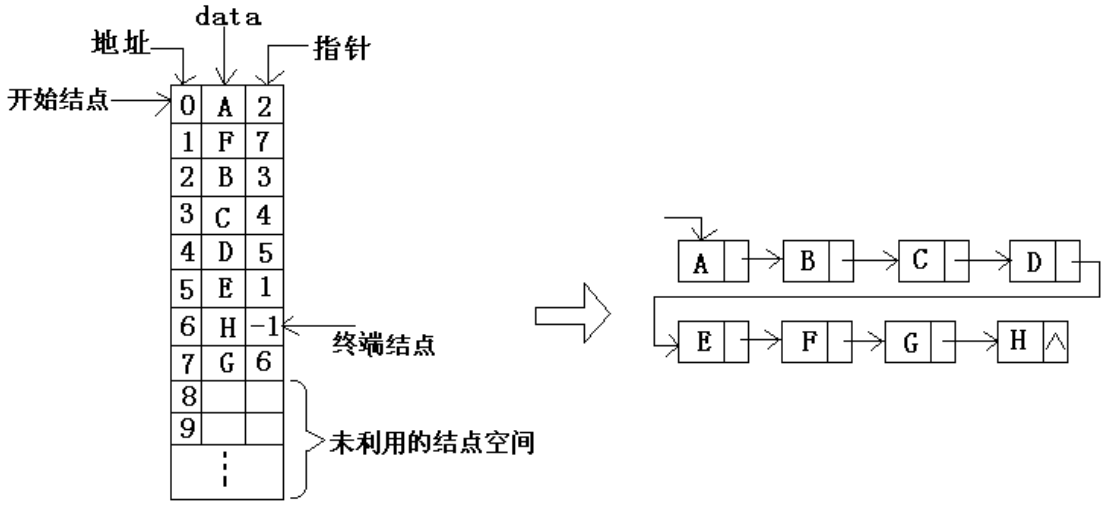


图 2.6 静态链表的表示

图 2.6 中左图是静态链表，右图是其对应的一般链表。一般链表结点空间是来自于整个内存，静态链表则来自于一个结构体数组，数组中每一个结点含有两个分量，一个是数据元素分量 data，另一个是指针分量，指示了当前结点的直接后继结点在数组中的位置（这和一般链表中 next 指针的地位是同等的）。

说明：在考研中经常要考到顺序表和链表的比较，这里给出一个较为全面的答案：

（1）基于空间的比较：

存储分配的方式：

顺序表的存储空间是静态分配的。

链表的存储空间是动态分配的。

存储密度(存储密度= 结点数据域所占的存储量/结点结构所占的存储总量)：

顺序表的存储密度 =1。

链表的存储密度<1（因为结点中有指针域）。

（2）基于时间的比较：

存取方式：

顺序表可以随机存取，也可以顺序存取（对于顺序表一般只答随机存取即可）

链表是顺序存取的。

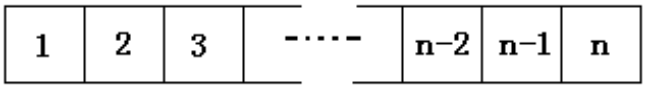
插入/删除时移动元素个数:

顺序表平均需要移动近一半元素。

链表不需要移动元素，只需要修改指针。

对顺序表进行插入和删除算法时间复杂度分析:

具有 n 个元素的顺序表（如下图），插入一个元素所进行的平均移动个数为多少（假设新元素插入在表中每个元素之后）。



因为题目要计算平均移动个数，这就是告诉我们要计算移动个数的期望。对于本题要计算期望的话就要知道在所有可能的位置插入元素时所对应的元素移动个数以及在每个位置发生插入操作的概率。

1 求概率:

因为插入位置的选择是随机的，所以所有位置被插入的可能性都是相同的，有 n 个可插入位置，所以任何一个位置被插入元素的概率都为 p=1/n。

2 求对应于每个插位置需要移动的元素个数:

假设要把新元素插入在表中第 i 个元素之后，则需要将 i 元素之后的所有元素往后移动一个位置，因此移动元素个数为 n-i。

由 1 和 2 知，移动元素个数的期望为:

$$p \times \sum_{i=1}^{n} n-i = \frac{n-1}{2}$$

即要移动近一半元素，由此即可以知道，插入和删除算法的平均时间复杂度为 O(n)。

2．2 线性表的基本操作

2．2．1 线性表的定义

```
#define MAX 100        //这里定义一个整型常量 MAX，值为 100。
```

1.顺序表的结构定义

```
typedef struct
{
    int  data[MAX];    //存放顺序表元素的数组（默认是 int 型，可根据题目要
                       //求将 int 换成其他类型）。
    int length;        //存放顺序表的长度。
}Sqlist;               //顺序表类型的定义。
```

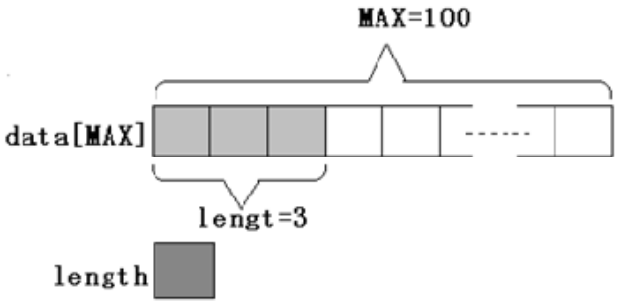


图 2.7 顺序表

如图 2.7 所示，一个顺序表包括一个存储表中元素的数组 data[]，和一个指示元素个数的变量 length。

说明：在考试中用的最多的顺序表的定义并不是这里讲到的结构型定义，而是如下这种形式：

```
int A[MAX];
int n;
```

上边这两句就定义了一个长度为 n，表内元素为整数的线性表。显然在答卷的时候这种定义方法要比定义结构体简洁一些。

2.单链表结点定义

```
typedef struct LNode
{
  int  data;            //data 中存放结点数据域（默认是 int 型）。
  struct LNode *next;   //指向后继结点的指针。
}LNode;                 //定义单链表结点类型。
```

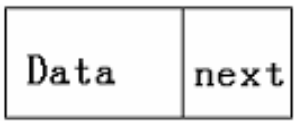


图 2.8 单链表结点结构图

3.双链表结点定义

```
typedef struct DLNode
{
  int  data;                 //data 中存放结点数据域（默认是 int 型）
  struct DLNode  *prior;     //指向后继结点的指针
  struct DLNode  *next;      //指向前驱结点的指针
}DLNode;                     //定义单链表结点类型
```

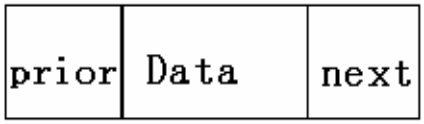


图 2.9 双链表结点结构图

说明：以上就是本章所需所有数据结构的 C 语言描述，希望考生牢记并且可以默写，这是完成本章程序设计题目所必须的最基本的东西。

2．2．2 顺序表的算法操作

说明：这一部分我们采取先讲例题，然后从例题中总结出这部分考研所的需基知识点。

例题 1 已知一个顺序表 L，其中的元素递增有序排列，设计一个算法插入一个元素 x（x 为 int 型）后保持该顺序表仍然递增有序排列（假设插入操作总能成功，即插入后表长不会大于 MAX）。

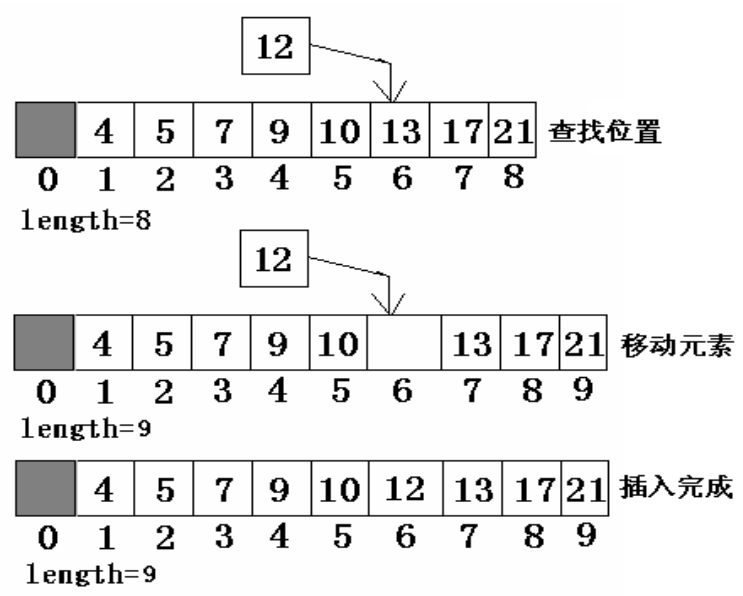
分析：

由题干可知，解决本题需完成两个操作。

一：要找出可以让顺序表保持有序的插入位置。

二：将第一步中找出的位置上以及其后的元素往后移动一个位置，然后将 x 放入腾出的位置上。

其执行过程图下：



元素插入过程图

说明：本书中如果不做特殊说明则默认数组元素的存放从下标 1 开始，0 号空位置不存数据以方便操作。

操作一：因为顺序表 L 中的元素是递增排列，所以我们可以从小到大逐个扫描表中元素，当找到第一个比 x 大的元素时，将 x 插入在这个元素之前即可。如上图所示，12 为要插入元素，从左往右逐个进行比较，当扫描到 13 的时候发现 13 是第一个比 12 大的数，因此 12 应插入在 13 之前。

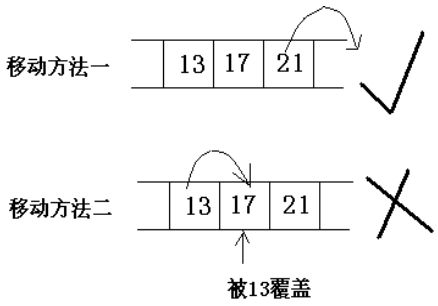
由此我们可以写出以下函数，本函数返回第一个比 x 大的元素的位置：

```
int LocateElem (Sqlist L,int x)
{
    int i;
    for(i=1;i<=L.length;i++)
        if(x<L.data[i])    //对顺序表中的元素从小到大逐个进行判
        {                  //断，看 x 是否小于当前所扫描到的元素，
            return i;      //如果小于则返回当前位置 i
        }
    return i;          //如果顺序表中不存在比 x 大的元素，则应
```

```
                                    //将 x 插入表尾元素之后，返回 i 来标记这
                                    //种情况（因 i<=L.length 这一句不成立
                                    //而退出 for 循环后 i 正好指示了表尾元素
                                    //之后的位置，同样也是正确的插入位置）。
}
```

操作二：

找到插入位置之后，将插入位置及其以后的元素向后移动一个元素的位置即可，这里有两种移动方法。一：先移动最右边的元素。二：先移动最左边的元素。哪个才是正确的移动方法呢？答案是先移动最右边的元素，如果先移动最左边的元素则右边的元素会被左边的元素所覆盖。如下图所示：



由此可以写出如下代码：

```
void insert(Sqlist &L,int x)  //因为 L 本身要发生改变，所以要用引用型。
{
    int p,i;
    p=LocateElem(L,x);        //调用 LocateElem()函数来找到插入位置 p。
    for(i=L.length;i>=p;i--)
       L.data[i+1]=L.data[i]; //从右往左逐个将元素右移一个位置。
    L.data[p]=x;              //将 x 放在插入位置 p 上。
    L.length++;               //表内元素多了一个，因此表长自增 1。
}
```

本题体现了考研中顺序表算法部分要求掌握的以下两个知识点：

（1）按元素值的查找算法。

在顺序表中查找第一个元素值等于 e 的元素（与上题中查找第一个比 x 大的元素是同样的道理），并返回其下标，代码如下。

```
int LocateElem (Sqlist L,int e)
{
    int i;
    for(i=1; i<=L.length;i++)
        if(e==L.data[i])
        return i;     //找到返回下表（下表均大于 0）。
    return 0;         //没找到返回 0，作为标记，因为 0 位置上不存放元素。
}
```

（2）插入数据元素的算法

在顺序表 L 的第 p（1≤p≤length+1）个位置上插入新的元素 e。如果 p 的输入不正确，则返回 0，代表插入失败；如果 p 的输入正确则将顺序表第 p 个元素及以后元素右移一个位置，腾出一个空位置插入新元素，顺序表长度增加 1，插入操作成功，返回 1。

插入操作代码如下：

```
int insert(Sqlist &L,int p,int e)
{
    int i;
    if(p<1||p>L.length+1||L.length==MAX)//位置错误或者表长已经达到
        return 0;                  //顺序表的最大允许值，此时插入不成功，返回0。
        for(i=L.length;i>=p;i--)
            L.data[i+1]=L.data[i];//从后往前逐个将元素往后移动一个位置。
        L.data[p]=e;               //将 x 放在插入位置 p 上。
        L.length++;                 //表内元素多了一个，因此表长自增 1。
        return 1;                  //插入成功返回 1。
}
```

例题 2 删除顺序表 L 中下标为 p（1≤p≤length）的元素，成功返回 1，否则返回 0，并将被删除元素的值赋值给 e。

分析：

要删除表中下标为 p 的元素，只需将其后边的元素逐个往前移动一个位置，将 p 位置上的元素覆盖掉，就达到了删除的目的。明白了上述元素插入的算法，本算法写起来就相对容易。只需将插入操作中的元素右移改成元素左移即可，右移的时候需要从最右边的元素开始移动，这里很自然想到左移的时候需要从最左边的元素开始移动。

由此可以写出以下代码：

```
int listDelete(Sqlist &L,int p,int &e)//需要改变的变量用引用型。
{
    int i;
    if(p<1||p>L.length)return 0;      //位置不对返回 0,代表删除不成功。
    e=L.data[p];                  //将被删除元素赋值给 e。
    for(i=p;i<=L.length;i++)    //从p位置开始将其后边的元素逐个前移一个位置。
        L.data[i]=L.data[i+1];
    L.length--;                    //表长减 1。
    return 1;                     //删除成功返回 1。
}
```

说明：通过以上两个例题，我们可以总结出顺序表的查找，插入和删除三种算法操作。这是考研中的重点，是考生必须熟练掌握的。

顺序表中还剩下两个比较简单的算法操作在这里稍微一提：

（1）初始化顺序表的算法：

只需将 length 设置为 0。

```
void InitList(Sqlist &L)  //L 本身要发生改变所以用引用型
{
     L.length=0;
}
```

（2）求指定位置元素的算法

用 e 返回 L 中 p（1≤p≤length）位置上的元素

```
int GetElem（Sqlist L,int p,int &e）//要改变所以用引用型
{
   if(p<1||p>L.length)             //p 值越界错误，返回 0
```

```
        return 0;
        e=L.data[p];
        return 1;
    }
```

2．2．3 单链表的算法操作（本书中如果没有特殊说明则链表都是含有头结点的链表）

例题 3 A 和 B 是两个单链表（带表头结点），其中元素递增有序。设计一个算法将 A 和 B 归并成一个按元素递增有序的链表 C，C 由 A 和 B 中的结点组成。

分析：

已知 A，B 中的元素递增有序，怎样使归并后的 C 中元素依然有序呢。我们可以从 A，B 中挑出最小的元素插入 C 的尾部，这样当 A，B 中所有元素都插入 C 中的时候，C 一定是递增有序的。哪一个元素是 A，B 中最小的元素呢？很明显，由于 A，B 是递增的，所以 A 中的最小元素是其开始结点中的元素，B 也一样。我们只需从 A，B 的开始结点中选出一个较小的来插入 C 的尾部即可。这里还需注意，A 与 B 中的元素有可能一个已经全部被插入到 C 中，另一个还没有插完，比如 A 中所有元素已经全部被插入到 C 中而 B 还没有插完，这说明 B 中所有元素都大于 C 中元素，因此只要将 B 链接到 C 的尾部即可，如果 A 没有插完则用类似的方法来解决。

经过以上分析我们可以写出如下代码：

```
    void merge(LNode *&A,LNode *&B,LNode *&C)
    {
        LNode *p=A->next; //p 来跟踪 A 的最小值结点。
        LNode *q=B->next; //q 来跟踪 B 的最小值结点。
        LNode *r;         //r 始终指向 C 的终端结点。
        C=A;              //用 A 的头结点来做 C 的头结点。
        C->next=NULL;
        free(B);          //B 的头结点已无用，则释放掉。
        r=C;              //r 指向 C，因为此时头结点也是终端结点。
        while(p!=NULL&&q!=NULL) //当 p 与 q 都不空时选取 p 与 q 所指结点中的较小
        {                       //者插入 C 的尾部。
          /*以下的 if  else 语句中，r 始终指向当前链表的终端节点，作为接纳新结点的一个媒介，通过它新结点被链接入 C 并且重新指向新的终端结点以便于接受下一个新结点，这里体现了建立链表的尾插法思想。*/
            if(p->data<=q->data)
            {
                r->next=p; p=p->next;
                r=r->next;
            }
            else
            {
                r->next=q; q=q->next;
                r=r->next;
            }
        }
        r->next=NULL;
  /*以下两个 if 语句将还有剩余结点的链表链接在 C 的尾部*/
        if(p!=NULL)
           r->next=p;
        if(q!=NULL)
           r->next=q;
    }
```

知识点总结：

例题.3 中涵盖了两个知识点，一是尾插法建立单链表，二是单链表的归并操作。上边的程序就是单链表归并操作的标准写法，希望同学们熟练掌握，下边我提取出尾插法建立单链表的算法，具体如下。

尾插法建立单链表：

假设有 n 个元素已经存储在数组 a 中，用尾插法建立链表 C。

```
void CreatelistR(LNode *&C,int a[],int n) //要改变的变量用引用型。
{
    LNode *s,*r;  //s 用来指向新申请的结点， r 始终指向 C 的终端结点。
    int i;
    C=(LNode *)malloc(sizeof(LNode));//申请 C 的头结点空间。
    C->next=NULL;
    r=C;                  //r 指向头结点，因为此时头结点就是终端结点。
    for(i=1;i<=n;i++)  //循环申请 n 个结点来接受数组 a 中的元素。
    {
        s=(LNode*)malloc(sizeof(LNode)); //s 指向新申请的结点。
        s->data=a[i];   //用新申请的结点来接受 a 中的一个元素 。
        r->next=s;     //用 r 来接纳新结点。
        r=r->next;     //r 指向终端结，点以便于接纳下一个到来的结点。
    }
    r->next=NULL;   //数组 a 中所有的元素都已经装入链表 C 中，C 的终端结点的指
                    //针域置为 NULL，C 建立完成。
}
```

以上是尾插法，与尾插法对应的建立链表的算法是头插法，代码如下：

```
void CreatelistF(LNode *&C,int a[],int n)
{
     LNode *s;
     int i;
     C=(LNode*)malloc(sizeof(LNode));
     C->next=NULL;
     for(i=1;i<=n;i++)
     {
       s=(LNode*)malloc(sizeof(LNode));
       s->data=a[i];
       /*下边两句是头插法的关键步骤。*/
       s->next=C->next;//s 所指新结点的指针域 next 指向 C 中的开始结点。
       C->next=s;//头结点的指针域 next 指向 s 结点,使得 s 成为了新的开始结点。
     }
}
```

在上述算法中不断的将新结点插入链表的前端,因此新建立的链表中元素的次序和数组 a 中的元素的次序是相反的。假如这里修改一下例题.3 的题干，将归并成一个递增的链表 C 改为归并成一个递减的链表 C。怎么来解决呢？答案是显然的，将插入过程改成头插法即可解决。代码如下，这里不需要 r 追踪 C 的终端结点，用 s 来接受新的结点插入链表 C 的前端。

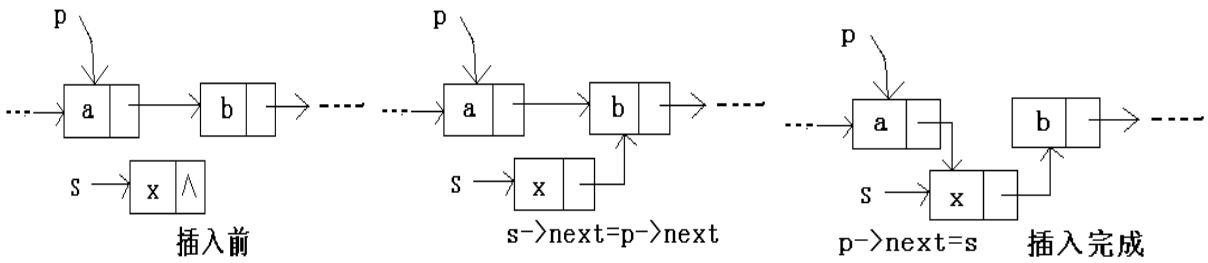
归并成递减的单链表算法：

```
void merge(LNode *&A,LNode *&B,LNode *&C)
 {
    LNode *p=A->next;
    LNode *q=B->next;
    LNode *s;
    C=A;
    C->next=NULL;
    free(B);
    while(p!=NULL&&q!=NULL)
    {  /*下边这个 if  else 语句体现了链表的头插法*/
        if(p->data<=q->data)
        {
           s=p;p=p->next;
           s->next=C->next;
           C->next=s;
        }
        else
        {
           s=q; q=q->next;
           s->next=C->next;
           C->next=s;
        }

    }
    /*下边这两个循是和求递增归并序列不同的地方，必须将剩余元素逐个插入 C 的头部才能得到最终的递减序列。*/
    while(p!=NULL)
    {
        s=p;p=p->next;
        s->next=C->next;
        C->next=s;
    }
    while(q!=NULL)
    {
        s=q;q=q->next;
        s->next=C->next;
        C->next=s;
    }
}
```

上边头插法的程序中提到了单链表的结点插入操作，此操作很简单，但有一点需要注意。假设 p 指向一个结点，要将 s 所指结点插入 p 所指结点之后的操作如下：

```
s->next=p->next;
p->next=s;
```

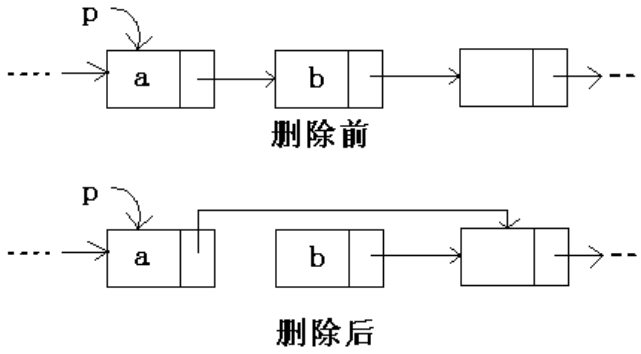


2.10 插入操作过程图

注意：以上插入操作语句能不能颠倒一下顺序写成 p->next=s;s->next=p->next;呢？显然是不可以的，因为第一句 p->next=s;虽然将 s 链接在 p 之后，但是同时也丢失了 p 直接后继结点的地址（p->next 指针原本所存储的 p 直接后继结点的地址在没有被转存到其他地方的情况下被 s 所覆盖，而正确的写法中，p->next 中的值在被覆盖前被转存在了 s->next 中，因而 p 后继结点的地址依然可以找到），这样链表断成了两截，没有满足将 s 插入链表的要求。

与插入结点对应的是删除结点，也比较简单，要将单链表的第 i 个结点删去，必须先在单链表中找到第 i-1 个结点，再删除其后继结点。如下图所示，若要删除结点 b，为了实现这一逻辑关系的变化，仅需要修改结点 a 中的指针域。假设 p 为指向 a 的指针。则只需将 p 的指针域 next 指向原来 p 的下一个结点的下一个结点即可。即

```
p->next=p->next->next;
```



2.11 删除操作过程图

这里还需注意，在考试答卷中，删除操作要释放所删除结点的内存空间，即完整的删除操作应该是这样：

```
q=p->next;
p->next=p->next->next;
free（q）;//调用 free 函数来释放 q 所指结点的内存空间。
```

掌握了单链表中结点删除的算法后，下边再看一个例题。

例题 2 查找链表 C（带头结点）中是否存在一个值为 x 的结点，存在就删除之并返回 1，否则返回 0。

分析：

对于本题需要解决两个问题，一是要找到值为 x 的结点；二是将找到的结点删除。问题一引出了本章要讲的单链表中最后一个重要操作，链表中结点的查找。为了实现查找，我们定义一个结点指针变量 p，让他沿着链表一直走到表尾，每来到一个新结点就检测其值是否为 x，是则证明找到，不是则继续检测下一个结点。

当找到值为 x 的结点后就是删除操作的内容，如何删除上面已经讲过。

由此可以写出以下代码：

```
int SearchAndDelete(LNode *&C,int x)
{
    LNode *p,*q;
    p=C;
   /*查找部分开始*/
    while(p->next!=NULL)
    {
        if(p->next->data==x)
           break;
        p=p->next;
    }
   /*查找部分结束*/
    if(p->next==NULL)
      return 0;
    else
    {
       /*删除部分开始*/
        q=p->next;
        p->next=p->next->next;
        free(q);
       /*删除部分结束*/
       return 1;
    }
}
```

说明：以上程序中之所以要使 p 指向所要删除结点的前驱结点而不是直接指向所要删除结点本身，是因为要删除一个结点必须知道其前驱结点的位置，这在之前删除操作的讲解中已经体现。

到此为止考研中对于顺序表和单链表算法操作部分所涉及的最重要的知识点都已经讲解完。考生务必要熟练掌握以上内容。下边要介绍的是双链表，循环链表，循环双链表的操作。这些内容考研中虽然也会涉及，常以选择题的形式出现，重要性不如以上两部分内容，并且这些内容是在上述两部分内容的基础上稍加变动而来的，容易理解。

2．2．4 双链表的算法操作

（1）采用尾插法建立双链表

```
void CreateDlistR(DLNode *&L,int a[],int n)
{
     DLNode *s,*r;
     int i;
     L=(DLNode*)malloc(sizeof(DLNode));
     L->next=NULL;
     r=L;  //和单链表一样 r 始终指向终端结点，开始头结点也是尾结点
     for(i=1;i<=n;i++)
     {
         s=(DLNode*)malloc(sizeof(DLNode));  //创建新结点
         s->data=a[i];
```

```
/*下边 3 句将 s 插入在 L 的尾部并且 r 指向 s，s->prior=r; 这一句是和建立单链表不同的地方。*/
            r->next=s;
            s->prior=r;
            r=s;
         }
         r->next=NULL;
    }
```

（2）查找结点的算法

在双链表中查找第一个结点值为 x 的结点。从第一个结点开始，边扫描边比较，若找到这样的结点，则返回结点指针，否则返回 NULL。算法代码如下：

```
DLNode* Finfnode(DLNode *C,int x)
{
    DLNode *p=C->next;
    while(p!=NULL)
    {
        if(p->data==x)
            break;
       p=p->next;
    }
    return p; //如果找到则 p 中内容是结点地址（循环因 break 结束），没找到
              //p 中内容是 NULL（循环因 p 等于 NULL 而结束）因此这一句可以
              //将题干中要求的两种返回值的种情况统一。
}
```

（3）插入结点的算法

假设在双链表中 p 所指的结点之后插入一个结点 s，其操作语句描述为：

```
s->next=p->next;
s->prior=p;
p->next=s;
s->next->prior=s;
```

指针变化过程如图 2.12 所示。

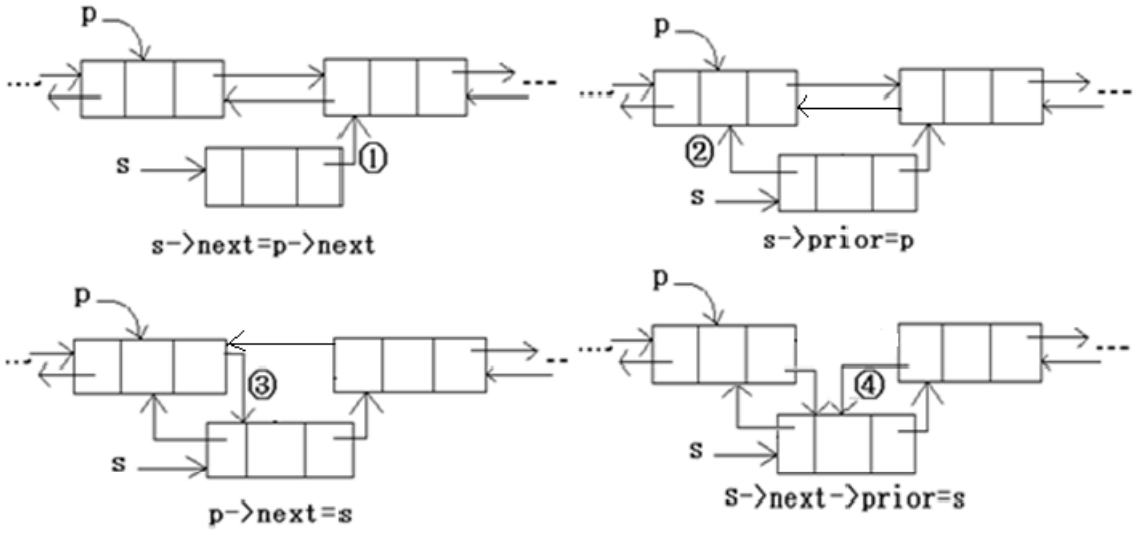


图 2.12 双链表结点插入过程图

说明：不知道大家有没有注意到在插入时，如果按照上面的顺序来插入，可以看成是一个万能的插入方式。不管怎样，先将要插入的结点两边链接好，这样有什么好处？对了，就是可以保证不会发生链断之后找不到结点的情况。所以请考生们一定要记住这种万能插

入结点的方式。

（4）删除结点的算法，设要删除双链表中 p 结点的后继结点，其操作的语句为：

```
q=p->next;
p->next=q->next;
q->next->prior=p;
free(q);
```

指针变化过程如图 2.13 所示。

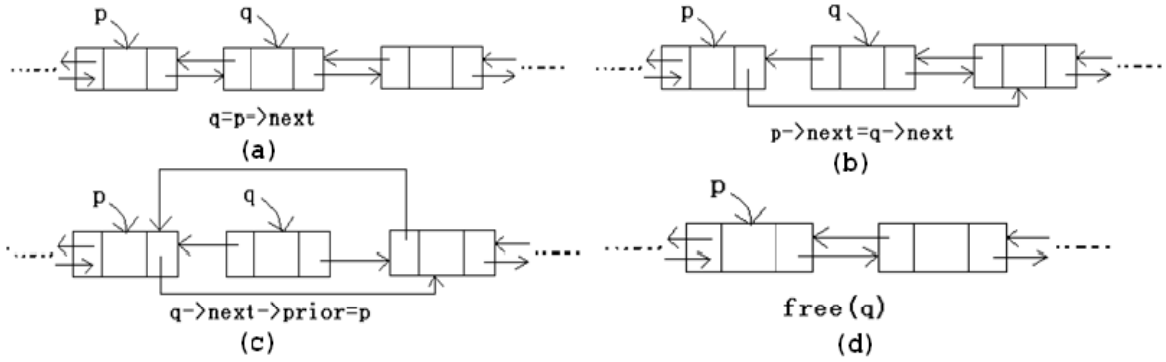


图 2.13 双链表结点删除过程图

2．2．5 循环链表的算法操作

循环单链表和循环双链表由对应的单链表和双链表改造而来，只需在终端结点和头结点间建立联系即可。循环单链表终端结点的 next 结点指针指向表头结点；循环双链表终端结点的 next 指针指向表头结点，头结点的 prior 指针指向表尾结点。需要注意的是如果 p 指针沿着循环链表行走，判断 p 走到表尾结点的条件是 p->next==head。循环链表的各种操作均与非循环链表类似，这里不再细说。

▲真题仿造(根据考研新大纲综合应用题的走向编制)

1.设顺序表用数组 A[]表示，表中元素存储在数组下标 1~m+n 的范围内，前 m 个元素递增有序，后 n 个元素递增有序，设计一个算法，使得整个顺序表有序。

（1）给出算法的基本设计思想。

（2）根据设计思想，采用C或C++语言描述算法，关键之处给出注释。

（3）说明你所设计算法的时间复杂度和空间复杂度。

2.已知递增有序的单链表 A,B（A,B 中元素个数分别为 m,n 且 A,B 都带有头结点）分别存储了一个集合，请设计算法以求出两个集合 A 和 B 的差集 A-B（即仅由在 A 中出现而不在 B 中出现的元素所构成的集合）。将差集保存在单链表 A 中，并保持元素的递增有序性。

（1）给出算法的基本设计思想。

（2）根据设计思想，采用C或C++语言描述算法，关键之处给出注释。

（3）说明你所设计算法的时间复杂度。

真题仿造答案与讲解：

1.解：

（1）算法基本设计思想：

将数组 A[]中的 m+n 个元素（假设元素为 int 型）看成两个顺序表，表 L 和表 R。将数组当前状态看做起始状态，即此时表 L 由 A[]中前 m 个元素构成，表 R 由 A[]中后 n 个元素构成。要使 A[]中 m+n 个元素整体有序只需将表 R 中的元素逐个插入表 L 中的合适位置即可。

插入过程：取表 R 中的第一个元素 A[m+1]存入辅助变量 temp 中，让 temp 逐个与 A[m],A[m-1],…,A[1]进行比较，当 temp<A[j]（1≤j≤m）时，将 A[j]后移一位；否则将 temp 存入 A[j+1]中。重复上述过程继续插入 A[m+2],A[m+3],……,A[m+n]，最终 A[]中元素整体有序。

（2）算法描述

```
void Insert(int A[],int m,int n)
{
    int i,j;
    int temp;                  //辅助变量，用来暂存待插入元素。
    for(i=m+1;i<=m+n;i++)  //将A[m+1…m+n]插入到A[1…m]中。
    {
        temp=A[i];
        for(j=i-1;j>=1&&temp<A[j];j--)
            A[j+1]=A[j];     //元素后移，以便腾出一个位置插入 temp。
        A[j+1]=temp;         //在 j+1 位置插入 temp。
    }
}
```

（3）算法时间和空间复杂度

①本题的规模由 m 和 n 共同决定。取最内侧循环中 A[j+1]=A[j];这一句作为基本操作，其执行次数在最坏的情况下为：

$f(m,n)=(m+m+n-1)n/2=mn+n^2/2-n/2$

由此可见本算法的时间复杂度为 $O(n^2+mn)$。

②算法额外空间中只有一个变量 temp，因此空间复杂度为 O(1)。

2.解

（1）算法基本设计思想：

只需从 A 中删去 A 与 B 中共有的元素即可。由于两个链表中元素是递增有序的所以可以这么做：设置两个指针 p,q 开始时分别指向 A 和 B 的开始结点。循环进行以下判断和操作，如果 p 所指结点的值小于 q 所指结点值，则 p 后移一位；如果 q 所指结点的值小于 p 所指结点的值，则 q 后移一位；如果两者所指结点的值相同，则删除 p 所指结点。最后 p 与 q 任一指针为 NULL 的时候算法结束。

（2）算法描述：

```
void  Difference(LNode *&A,LNode *B)
{
    LNode *p=A->next,*q=B->next;    //p和q分别是链表A和B的工作指针。
    LNode *pre=A;                   //pre为A中p所指结点的前驱结点的指针。
    LNode *r;
    while(p!=NULL&&q!=NULL)
    {
        if(p->data<q->data)
        {
            pre=p;
            p=p->next;           //A链表中当前结点指针后移。
        }
        else if(p->data>q->data)
            q=q->next;           //B链表中当前结点指针后移。
        else
        {
            pre->next=p->next;    //处理A,B中元素值相同的结点,应删除。
            r=p;
            p=p->next;
            free(r);              //删除结点。
         }
    }
}
```

（3）算法时间复杂度分析：

由算法描述可知，算法规模由 m 和 n 共同确定。算法中有一个单层循环，循环内的所有操作都是常数级的，因此可以用循环执行的次数作为基本操作执行的次数。可见循环执行的次数即为 p,q 两指针沿着各自链表移动的次数，考虑最坏的情况，即 p,q 都走完了自己所在的链表，循环执行 m+n 次。即时间复杂度为 O(m+n)。

习题心选

一 .选择题（题目中的链表如果不特殊指出就是带头结点的链表）

1．下述哪一条是顺序存储结构的优点？（ ）

A．存储密度大　　B．插入运算方便

C．删除运算方便　　D．可方便地用于各种逻辑结构的存储表示

2．下面关于线性表的叙述中，错误的是哪一个？（ ）

A．线性表采用顺序存储，必须占用一片连续的存储单元。

B．线性表采用顺序存储，便于进行插入和删除操作。

C．线性表采用链接存储，不必占用一片连续的存储单元。

D．线性表采用链接存储，便于插入和删除操作。

3．线性表是具有 n 个（ ）的有限序列。

A．表元素　　B．字符　　C．数据元素　　D．数据项

4．若某线性表最常用的操作是存取任一指定序号的元素和在最后进行插入和删除运算，则利用（　）存储方式最节省时间。

A．顺序表　　B．双链表　　C．双循环链表　　D．单循环链表

5．某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则采用（　）存储方式最节省运算时间。

A．单链表　　B．不带头结点的单循环链表

C．双链表　　D．不带头结点且有尾指针的单循环链表

8．静态链表中指针指示的是（　）。

A．内存地址　　B．数组下标

C．链表中下一元素在数组中的地址　　D．左、右孩子地址

9．链表不具有的特点是（　）。

A．插入、删除不需要移动元素

B．可随机访问任一元素

C．不必事先估计存储空间

D．所需空间与线性长度成正比

10.将两个有 n 个元素的有序表归并成一个有序表，其最少比较次数为（　）。

A.n　　B.2n-1　　C.2n　　D.n-1

11.单链表 L（带头结点）为空的判断条件是（　）。

A. L==NULL　　B. L->next==NULL　　C. L->next!=NULL　　D. L!=NULL

12 在一个具有 n 个结点的有序单链表中插入一个新结点并仍然保持有序的时间复杂度是（　）。

A. O(1)　　B. O(n)　　C.O(n^2)　　D. O($n\log_2 n$)

13.在一个长度为 n（n>1）的带头结点的单链表 h 上，另设有尾指针 r（指向尾结点），执行（　）操作与链表的长度有关。

A．删除单链表中的第一个结点。

B．删除单链表中的最后一个结点。

C．在单链表第一个元素前插入一个新结点。

D．在单链表最后一个元素后插入一个新结点。

14.在一个双链表中，在 p 结点之后插入结点 q 的操作是（　）。

A. q->prior=p; p->next=q; p->next->prior=q; q->next=p->next;

B. q->next=p->next; p->next->prior=q; p->next=q; q->prior=p;

C. p->next=q; q->prior=p; q->next=p->next; p->next->prior=q;

D. q->prior=p; p->next=q; q->next=p->next; p->next->prior=q;

15.在一个双链表中，在 p 结点之前插入 q 结点的操作是（　）。

A. p->prior=q; q->next=p; p->prior->next=q; q->prior=p->prior;

B. q->prior=p->prior;　　p->prior->next=q;
　　q->next=p;　　p->prior=q->next;

C. q->next=p; p->next=q; q->prior->next=q; q->next=p;

D. p->prior->next=q; q->next=p; q->prior=p->prior;p->prior=q;

16.在一个双链表中，删除 p 结点的操作是（ ）（结点空间释放语句省略）。

A. p->prior->next=p->next; p->next->prior=p->prior;

B. p->prior=p->prior->prior; p->prior->prior=p;

C. p->next->prior=p; p->next=p->next->next;

D. p->next=p->prior->prior; p->prior=p->prior->prior;

17.非空的单循环链（带头结点）表 L 的终端结点（由 p 所指向）满足（ ）。

A.p->next==NULL B.p==L

C.p->next==L D.p->next==L&&p!=L

18. 带头结点的双循环链表 L 为空的条件是（ ）。

A. L==NULL B. L->next->prior==NULL

C. L->prior==NULL D. L->prior==L||L->next==L

19. 线性表是（ ）。

A. 一个有限序列，可以为空。 B. 一个有限序列，不可以为空。

C. 一个无限序列，可以为空。 D. 一个无限序列，不可以为空。

20.线性表采用链表存储时，其地址（ ）。

A.必须是连续的。 B.一定是不连续的。

C.部分地址必须是连续的。 D.连续与否均可以。

21.线性表的静态链表存储结构，与顺序存储结构相比其优点是（ ）。

A.所有的操作算法实现简单。 B.便于随机存取。

C.便于插入和删除。 D.便于利用零散的存储空间。

22.设线性表有 n 个元素，以下操作中，（ ）在顺序表上实现比在链表上实现效率更高。

A.输出第 i(1≤i≤n)个元素值。

B.交换第 1 个元素与第 2 个元素的值。

C.顺序输出这 n 个元素的值。

D.输出与给定值 x 相等的元素在线性表中的序号。

23.对于一个线性表，既要求能够快速的进行插入和删除，又要求存储结构能够反映数据元素之间的逻辑关系，则应采用（ ）存储结构。

A.顺序 B.链式 C.散列（Hash 表）

24.需要分配较大的空间,插入和删除不需要移动元素的线性表,其存储结构为( )。

A.单链表 B.静态链表 C.顺序表 D.双链表。

25.如果最常用的操作是取第 i 个元素的前驱结点，则采用（ ）存储方式最节省时间。

A.单链表 B.双链表 C.单循环链表 D.顺序表

26.与单链表相比，双链表的优点之一是（ ）。

A.插入、删除操作更简单。 B.可以进行随机访问。

C.可以省略表头指针或表尾指针。 D.访问前后相邻结点更灵活。

27.在顺序表中插入一个元素的时间复杂度为（ ）。

A.O(1) B.O($\log_2 n$) C.o(n) D.o(n^2)

28.在顺序表中删除一个元素的时间复杂度为（ ）。

A.O(1) B.O($\log_2 n$) C.o(n) D.o(n^2)

29.对于一个具有 n 个元素的线性表，建立其单链表的时间复杂度为（ ）。

A.O(1) B.O($\log_2 n$) C.o(n) D.o(n^2)

30.若某表最常用的操作是在最后一个结点之后插入一个结点或删除最后一个结点，则采用（ ）存储结构最节省运算时间。

A.单链表。 B.给出表头指针的循环单链表。

C.双链表。 D.带头结点的循环双链表。

31.线性表最常用的操作是在最后一个结点之后插入一个结点或删除第一个结点，则采用（ ）存储方式最节省时间。

A.单链表 B.仅有头结点的单循环链表

C.双链表 D.仅有尾结点指针的单循环链表。

32.设有两个长度为 n 的单链表（带头结点），结点类型相同，若以 h1 为头结点指针的链表是非循环的，以 h2 为头结点指针的链表是循环的，则（ ）。

A.对于两个链表来说，删除开始结点的操作，其时间复杂度分别为 O(1)和 O(n)。

B.对于两个链表来说，删除终端结点的操作，其时间复杂度都是 O(n)。

C.循环链表要比非循环链表占用更多的内存空间。

D.h1 和 h2 是不同类型的变量。

二：综合应用题

（1）基础题

1.设计一个算法，将顺序表中的所有元素逆置。

2.设计一个算法，从一给定的顺序表 L 中删除下标 i 到 j（i≤j，包括 i，j）之间的所有元素，假定 i,j 都是合法的。

3.有一个顺序表 L，其元素为整型数据，设计一个算法，将 L 中所有小于表头元素的整数放在前半部分，大于的整数放在后半部分，数组从下表 1 开始存储。

4.有一个递增非空单链表，设计一个算法删除值域重复的结点。比如{1,1,2,3,3,3,4,4,7,7,7,9,9,9}经过删除后变成{1,2,3,4,7,9}。

5.设计一个算法删除单链表 L（有头结点）中的一个最小值结点。

6.有一个线性表，采用带头结点的单链表 L 来存储。设计一个算法将其逆置。要求不能建立新结点，只能通过表中已有结点的重新组合来完成。

7.设计一个算法将一个头结点为 A 的单链表（其数据域为整数）分解成两个单链表 A

和 B,使得 A 链表只含有原来链表中 data 域为奇数的结点,而 B 链表只含有原链表中 data 域为偶数的结点，且保持原来相对顺序。

（2）思考题

1.有 N 个个位正整数存在 int 型数组 A[0……N-1]中,N 为已定义好的常量且 N≤9，数组 A[]的长度为 N，另给一个 int 型变量 i，要求只用上述变量（即 A[0]~A[n-1]与 i 这 N+1 个整型变量）写算法找出这 N 个整数中的最小者，并且要求不能破坏数组 A[]中的数据。

2.写一个函数，逆序打印单链表中的数据，假设指针 L 指向了单链表的开始结点。

习题心讲

选择题：

1.A

本题考查顺序表与链表的对比。顺序表不像链表一样在结点中存在指针域，因此存储密度大，A 正确。BC 两项是链表的优点。D 中可方便地用于各种逻辑结构的存储表示是错误的，比如对于树型结构，顺序表显然不如链表表示起来更方便。

2.B

本题考查线性表两种存储结构的特点。线性表的顺序存储结构必须占用一片连续的存储空间，而链表不需要这样，这在顺序存储结构中已经讲过。顺序表不便于元素的插入和删除，因为要移动多个元素。链表在插入和删除的过程中不需要移动元素，只需修改指针。因此本题选 B

3.C

本题考查线性表的定义。线性表的定义：线性表是具有相同特性**数据元素**的一个有限序列。

4.A

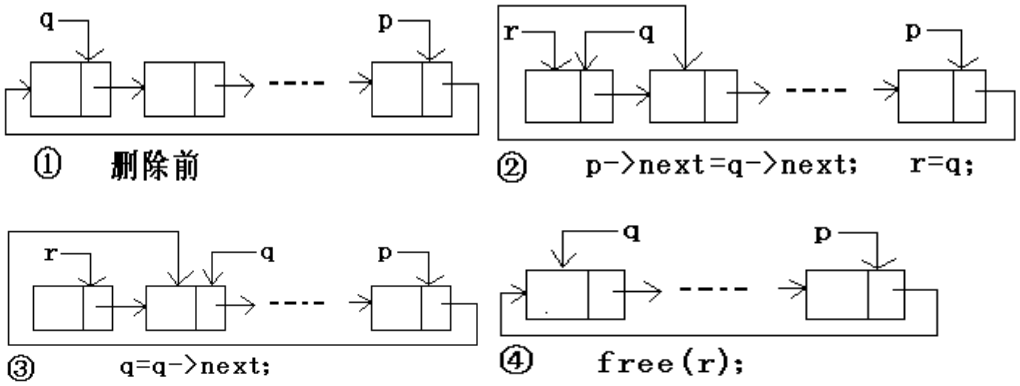
本题考查顺序表的特点。顺序表进行插入或者删除操作时需要移动的元素是待插或者待删位置之后的元素，因此当插入或者删除操作发生在顺序表的表尾时，不需要移动元素。顺序表支持随机存储,方便于存取任意指定序号的元素,因此本题情况下顺序表最节省时间。

5.D

本题考查几种链表的插入和删除操作。

对于 A 项，单链表，要在最后一个元素后插入一个元素，需要遍历整个链表才能找到插入位置时间复杂度是 O(n)。删除第一个元素时间复杂度是 O(1)。

对于 B 项，不带头结点的单循环链表在最后一个元素之后插入一个元素的情况和单链表相同，时间复杂度是 O(n)。对于删除表中第一个元素的情况，同样需要找到终端结点，即删除结点的时间复杂度也是 O(n)，因为终端结点的指针 next 指向开始结点，首先需要遍历整个链表找到终端结点,然后执行 p->next=q->next;r=q;q=q->next;free(r);四句才满足删除要求（其中 p 指向终端结点，q 指向起始结点，r 用来帮助释放结点空间）。删除过程示意图如下：



说明：本选项的讲解中顺便给出了不带头结点的单循环链表中删开始结点的算法过程，供考生参考。

对于 C 项，双链表的情况和单链表相同，一个 O(n)，一个 O(1)。

对于 D 项，有尾指针（即指向表中终端结点的指针），因为已经知道终端结点的位置所以省去了 B 选项中链表的遍历过程，因此删除和插入结点的时间复杂度为 O(1)。

综上所述，本题选 D

8.C

本题考查静态链表中指针作用。虽然静态链表用数组来存储，但其指针域和一般单链表一样，指出了表中下一个结点的位置。不要因为静态链表是用数组来存储就想当然的将静态链表等同于顺序表，而误选 B 项。

9.B

本题考查顺序表和链表的比较。显然随机存储是顺序表的特性，而不是链表的特性。

10.A

本题考查线性表的归并算法。在归并算法中，当 L1 表中的所有元素均小于 L2 表中的所有元素时，比较次数最少，为 n。因为此时 L1 中的元素在比较了 n 次后已经全部被并入顺序表，剩下的 L2 中的元素就不需要比较。

补充：如果本题问比较次数最多是几次该选什么呢。答案是 2n-1 次，此时 L1 与 L2 中元素轮流被并入顺序表，每并入一个就比较一次，当剩下最后 2 个元素的时候（L1 与 L2 表各一个），已经比较了 2n-2 次，这时还需进行一次比较就有一个表变为空表，至此所有比较结束，因此总的比较次数是 2n-1 次。

11.C

本题考查带头结点的单链表和不带头结点的单链表基本操作的区别。带头结点的单链表判空条件看 head->next 是否为 NULL，不带头结点的单链表是看头指针 head 是否为 NULL。

12.B

本题考查链表的查找与插入操作。这里假设单链表递增有序（递减的情况同理），在插入数据为 x 的结点之前，先要在单链表中找到第一个大于 x 的结点的直接前驱 p，在 p 之后插入该结点。查找的过程时间复杂度为 O(n)，插入过程的时间复杂度为 O(1)，整个过程时间复杂度为 O(n)。

13.B

本题考查链表的删除操作。在带有头结点的链表中要删除一个结点，必须知道其前驱

结点位置，由题干知，第一个结点的前驱结点位置已知，为 h 所指的头结点，因此无论链表多长删除其中第一个结点的操作都是p=h->next;h->next=p->next;free(p);这三条语句。尾结点的前驱结点地址未知，因此要删除链表中最后一个结点，需要从头结点开始遍历整个链表来找到最后一个结点的前驱结点，链表越长执行操作的循环次数越多，即删除最后一个结点的操作与链表长度有关。

要在某一个结点后插入一个结点，则必须知道这个结点的地址，C 项中在第一个元素前插入一个新结点，等价于在头结点后插入一个新结点，头结点地址已知，插入操作与链表长度无关。同样，对于 D 选项尾结点地址已知，在其后插入新结点的操作与链表长度无关。

14.B

本题考查双链表的插入操作。

参见之前双链表插入操作执行图可知本题选 B

A 选项执行完第二句时 p 的后继结点地址丢失，插入不能完成。

C 选项执行完第一句时 p 的后继结点地址丢失，插入不能完成。

D 选项执行完第二句时 p 的后继结点地址丢失，插入不能完成。

15.D

本题考查链表的插入操作。

A 选项执行完第一句时 p 的前驱结点地址丢失，插入不能完成。

B 选项最后一句 p->prior=q->next;应该改为 p->prior=q;

C 选项破坏了 p 和其后继结点的联系。

16.A

本题考查链表的插入操作。

参见之前双链表删除算法执行图可知本题选 A

B 选项第一句执行完之后 p 的后继结点和前边的所有结点都失去联系。

C 选项第一句为废操作，第二句执行完 p 的后继结点地址丢失。

D 选项将 p 结点的 next 指针指向了其前驱的前驱，不满足删除要求。

17.D

本题考查单循环链表的判空条件。

带头结点的单循环链表中没有空指针域，因此 A 不对。

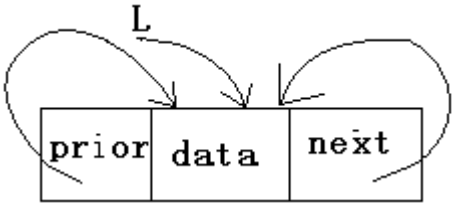
当终端结点与头结点为同一个结点，即满足 p==L 时，链表为空因此 B 不对。

当链表为空时，p 指向头结点，同时也是终端结点，此时一定满足 p->next==L。当链表不空时，p 指向终端结点，同样满足 p->next==L，因此 C 项恒成立，不能作为判断链表非空的条件。

因此本题答案为 D。

18.D

本题考查双循环链表的判空条件。链循双环链表为空时头结点体现如下形态：



可见当满足 L->prior==L||L->next==L 时，链表为空。并且循环双链表同循环单

链表一样，没有空指针域，因此本题选 D。

19.A

本题考查线性表的定义。线性表是具有相同特性数据元素的一个有限序列。该序列中所含元素的个数叫做线性表的长度，用 n 表示（n≥0）。注意 n 可以等于零，表示线性表是一个空表，空表也可以作为一个线性表。因此本题选 A。

20.D

本题考查链表存储结构的特点。一般链表结点在内存中是分散存在的，因此可是不连续的；当然如果连续的申请一片存储空间来存储链表结点也未尝不可。因此本题选 D。

21.C

本题考查静态链表与顺序表的特点。静态链表具有链表的插入和删除方便的特点，也不需要移动较多的元素，这是优于顺序表的地方，因此本题选 C。

22.A

本题考查顺序表与链表的特点。

顺序表支持随机存储，链表不支持，因此顺序表输出地 i 个元素值的时间复杂度为 O(1)，链表则为 O(n)，因此 A 正确。

交换第 1 与第 2 个元素的值，对于顺序表与链表，时间复杂度均为 O(1)，因此 B 不对。

输出 n 个元素的值，两者时间复杂度均为 O(n)，因此 C 不对。

输出与给定值 x 相等的元素在线性表中的序号，对于顺序表和链表，都需要搜索整个表，因此时间复杂度为 O(n)，D 不对。

说明：对于 B 项，写出两者的具体操作如下：

线性表：temp=a[1];a[1]=a[2];a[2]=temp;

要执行 3 次操作。

链表：

```
p=head->next->data;
q=head->next->next->data;
temp=p->data;
p->data=q->data;
q->data=p->data;
```

需要执行 5 次操作。因此严格来说，B 项也正确。在考研中这种情况很容易出现，考生遇到这种题目的时候要选择“更准确”的一项，显然比起 B 项来说，A 项更准确。

23.B

本题考查链表的特点。链表较之题选项中其他两种存储结构的最大优点就是能够快速的进行插入和删除，当然链表也能反映数据元素之间的逻辑关系，Hash 表中各元素之间不存在逻辑关系，因此本题选 B。

24.B

本题考查静态链表的特点。静态链表用数组来表示，因此需要分配较大的连续空间，静态链表同时还具有一般链表的特点，即插入和删除操作不需要移动元素。因此本题选 B。

25.D

本题考查顺序表的特点。较之其他三种存储结构，顺序表找到表中第 i 个元素的时间复杂度最小，为 O(1)；取第 i 个元素前驱的时间复杂度也是最小，为 O(1)。因此本题选

D。

26.D

本题考查双链表的特点。在双链表中可以快速访问任何一个结点的前后相邻结点，而单链表中只能快速访问任何一个结点的后继结点。因此本题选 D。

27.C

本题考查顺序表插入算法的效率。顺序表插入算法的基本操作是元素的移动。在一个长度为 n 的顺序表中的位置 i 上插入一个元素，需要进行元素移动的次数大约为 n-i 次，插入算法元素平均移动次数大约为 n/2 次，因此时间复杂度为 O(n)。

28.C

本题考查顺序表删除算法的效率。顺序表删除算法的基本操作是元素的移动。在一个长度为 n 的顺序表中的 i 位置上删除一个元素，需要进行元素移动的次数大约为 n-i 次，删除算法平均元素移动次数大约为 n/2 次，因此时间复杂度为 O(n)。

29.C

本题考查链表的建立。链表建立的过程即将结点逐个插入链表的过程，因此链表建立的时间复杂度即为链表规模 n 乘以链表插入操作的时间复杂度 O(1)，即 n×O(1)，即 O(n)。因此本题选 C。

30.D

本题考查链表的基本操作。在链表中插入或删除一个结点，需要修改相邻结点的指针域。如不特别指明，通常只给出链表头结点的地址，其他结点的地址只能从它的前驱或者后继得到。以上 4 种选项中只有 D 能通过头结点指针直接获得最后一个结点的相邻结点的地址，因此本题选 D。

31.D

本题考查链表的基本操作。在有尾结点指针 r 的循环单链表中，在最后一个结点之后插入结点 s 的操作是：s->next=r->next;r->next=s;r=s;删除第一个结点的操作是：p=r->next;r->next=p->next;free(p);其时间复杂度均为 O(1)。因此本题选 D。

32.B

本题考查链表的综合知识。

对于两个链表来说，要删除开始结点，其时间复杂度都为 O(1)。因为已知开始结点的前驱结点地址，即头结点地址。

对于两个链表来说，要删除终端结点，都需要从头结点开始找到终端结点的前驱结点，其时间复杂度都是 O(n)。

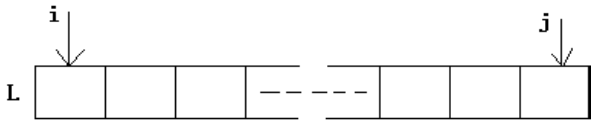
两链表占用内存空间相同。不要误以为循环链表比非循环链表多一个指向头结点的指针，其实非循环链表也有这个指针，只是它没有指向头结点而已。

h1 和 h2 指向相同类型的结点，因此是相同类型的变量。单链表和循环单链表结点类型相同，只是表中结点的组织方式不同。

综合应用题：

（1）基础题

1．分析：



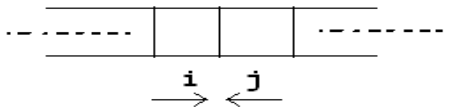
第 1 题图

本算法简单，上图即可说明问题，两个变量 i，j 指示顺序表的第一个元素和最后一个元素，交换 i，j 所指元素，然后 i 向后移动一个位置，j 向前移动一个位置，如此循环，直到 i 与 j 相遇时结束，此时顺序表 L 中的元素已经逆置。

由此可写出以下代码：

```
void reverse(Sqlist &L) //L 要改变，用引用型
{
    int i,j;
    int temp;  //辅助变量，用于交换
    for(i=1,j=L.length;i<j;i++,j--) //当 i 与 j 相遇时循环结束
   {
      temp=L.data[i];
      L.data[i]=L.data[j];
      L.data[j]=temp;
   }
}
```

注意：本题中 for 循环的执行条件要写成 i<j 而不要写成 i！=j。如果数组中元素有偶数个则 i 与 j 会出现下图所示状态，此时 i 继续往右走，j 继续往左走，会互相跨越对方，循环不会结束。



第 1 题图

2．分析：

本题是顺序表删除算法的扩展，可以采用如下方法解决，从第 j+1 个元素开始到最后一个元素为止，用这之间的每个元素去覆盖从这个元素开始往前数第 j-i+1 个元素，即可完成删除 i~j 之间的所有元素。

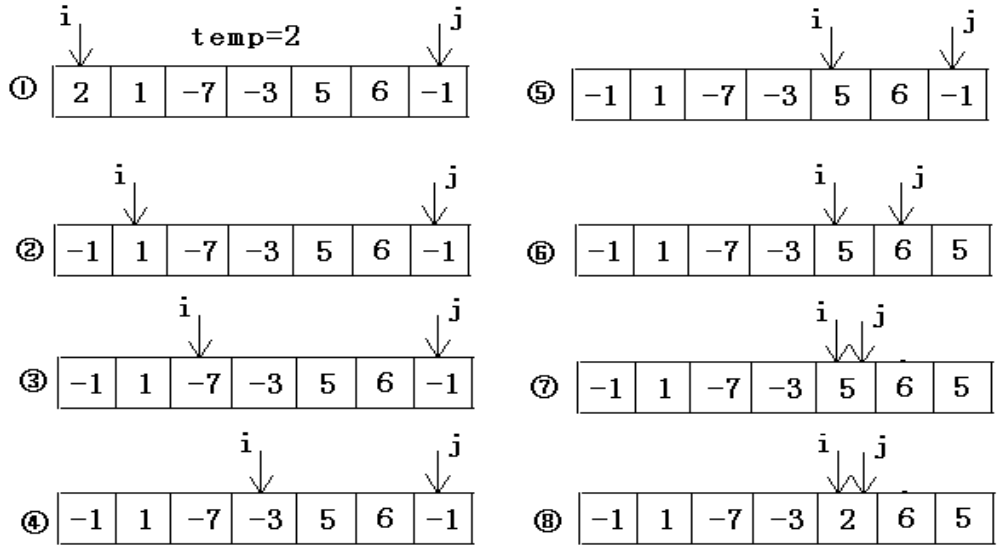
本题代码如下：

```
void Delete(Sqlist &L,int i,int j)//L 要改变，用引用型。
{
    int k,l;
    l=j-i+1;         //元素要移动的距离。
    for(k=j+1;k<=L.length;k++)
    {
        L.data[k- l]=L.data[k]; //用第 k 个元素去覆盖它前边的第 l 个元素。
    }
    L.length-=l;    //表长改变。
}
```

3．分析：

本题可以这样解决，先将 L 的第一个元素存于变量 temp 中，然后定义两个整型变量

i,j。i 从左往右扫描，j 从右往左扫描。边扫描边交换。具体执行过程如下：



第 3 题图

各步的解释如下：

① 开始状态，temp=2,i=1;j=L.length

② j 先移动，从右往左，边移动边检查 j 所指元素是否比 2 小，此时发现-1 比 2 小，则执行 L.data[i]=L.data[j];i++;(i 中元素已经被存入 temp 所以可以直接覆盖，并且 i 后移一位，准备开始 i 的扫描)

③ i 开始移动，从左往右，边移动边检测，看是否 i 所指元素比 2 大，此时发现-7 比 2 小，因此 i 在此位置是什么都不做。

④ i 继续往右移动，此时 i 所指元素为-3 也比 2 小，此时什么都不做。

⑤ i 继续往右移动,此时 i 所指元素为 5,比 2 大,因此执行 L.data[j]=L.data[i];j--（j 中元素已被保存，j 前移一位，准备开始 j 的扫描）

⑥ j 往左运动，此时 j 所指元素为 6，比 2 大，j 在此位置时，什么都不做。

⑦ j 继续往左移动，此时 j==i，说明扫描结束。

⑧ 执行 L.data[i]=temp;此时整个过程结束，所有比 2 小的元素被移到了 2 前边，所有比 2 大的元素被移到了 2 后边。

以上过程要搞清楚两点：

① i 和 j 是轮流移动的，即当 i 找到比 2 大的元素时，将 i 所指元素放入 j 所指位置，i 停在当前位置不动，j 开始移动。j 找到比 2 小的元素，将 j 所指元素放在 i 所指位置，j 停在当前位置不动，i 开始移动，如此交替直到 i==j。

② 每次元素覆盖（比如执行 L.data[i]=L.data[j];）不会造成元素丢失，因为在这之前被覆盖位置的元素已经存入其他位置。

由以上分析可写出如下算法：

```
void move(Sqlist &L)//L 要改变所以用引用型
{
    int temp;
    int i=1,j=L.length;
    temp=L.data[i];
    while(i<j)
    {
      /*关键步骤开始*/
while(i<j&&L.data[j]>temp) j--; //j 从左往右扫描，当来到第一个比 temp
                                //小的元素时停止,并且每走一步都要判
                                //断 i 是否小于 j,这个判断容易遗漏。
```

```
        if(i<j)                          //检测看是否已仍满足 i<j，这一步同样很
        {                                //重要有很多考生忘记这个判断。
            L.data[i]=L.data[j]; //移动元素。
            i++;                    //i 右移一位。
        }
        while(i<j&&L.data[i]<temp) i++;  //与上边的处理类似。
        if(i<j)                           //与上边的处理类似。
        {
            L.data[j]=L.data[i];          //与上边的处理类似。
            j--;                          //与上边的处理类似。
        }
           /*关键步骤结束*/
        }
        L.data[i]=temp;                  //将表首元素放在最终位置。
    }
```

说明：在这里之所以如此详细的讲解此题的执行过程，是因为此题是后边排序章节中快速排序的关键步骤，因此考生务必将其执行过程以及算法熟练掌握。

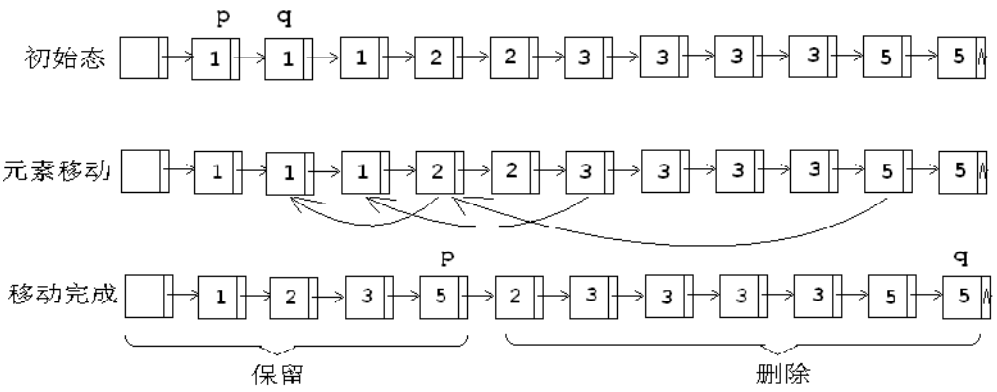
4．分析：

解法一：定义指针 p 指向起始结点。将 p 所指当前结点值域和其直接后继结点值域比较。如果当前结点值域等于后继结点值域，删除后继结点；否则 p 指向后继结点，重复以上过程，直到 p 的后继结点为空。

本题代码如下：

```
void delsl1(LNode *&L)             //L 是指针且要改变因此用引用指针型
{
    LNode *p=L->next,*q;
    while(p->next!=NULL)
    {
        if(p->data==p->next->data) //找到重复结点删除之
        {
            q=p->next;
            p->next=q->next;
            free(q);
        }
        else
            p=p->next;
    }
}
```

解法二：依次将原序列中每个连续相等子序列的第一个元素移动到表的前端，将剩余的元素删除即可，即下图所示过程。



第 4 题图

令 p 指向起始将结点。q 从 p 的后继结点开始扫描，q 每来到一个新结点的时候进行检测：当 q->data 等于 p->data 的时候，什么也不做，q 继续往后走；当两者不相等的时候，p 往后走一个位置，然后用 q->data 取代 p->data。之后 q 继续往后扫描，重复以上过程。当 q 为空的时候，释放从 p 之后的所有结点空间。

```
void delsl2(LNode *&L)
{
    LNode *p=L->next,*q=L->next->next,*r;
    while(q!=NULL)
    {
        while(q!=NULL&&q->data==p->data)
            q=q->next;
        if(q!=NULL)
        {
            p=p->next;
            p->data=q->data;
        }
    }
    q=p->next;
    p->next=NULL;
    while(q!=NULL)
    {
        r=q;
        q=q->next;
        free(r);
    }
}
```

5．分析：

用 p 从头至尾扫描链表，pre 指向*p 结点的前驱，用 minp 保存值最小的结点指针，minpre 指向 minp 的前驱。一边扫描，一边比较，将最小值结点放到 minp 中。

代码如下：

```
void delminnode(LNode *&L)
{
    LNode *pre=L,*p=pre->next,*minp=p,*minpre=pre;
    while(p!=NULL)            //查找最小值结点 minp 以及前驱结点 minpre
    {
        if(p->data<minp->data)
        {
            minp=p;
```

```
            minpre=pre;
        }
        pre=p;
        p=p->next;
    }
    minpre->next=minp->next;   //删除*minp 结点。
    free(minp);
}
```

6. 分析：

在前边讲过的算法基础中，提到过关与逆序的问题，那就是链表建立的头插法。头插法完成后，链表中的元素顺序和原数组中元素的顺序相反。这里我们可以将 L 中的元素作为逆转后 L 的元素来源，即将 L->next 设置为空，然后将头结点后的一串结点用头插法逐个插入 L 中，这样新的 L 中的元素顺序正好是逆序的。

本题代码如下：

```
void Reversel(LNode *&L)
{
    LNode *p=L->next,*q;
    L->next=NULL;
    while(p!=NULL)       //p 结点始终指向旧的链表的开始结点。
    {
        q=p->next;      //q 结点作为辅助结点来记录 p 的直接后继结点的位置。
        p->next=L->next;  //将 p 所指结点插入新的链表中。
        L->next=p;
        p=q;            //因为后继结点已经存入 q 中，因此 p 仍然可以找到后继
                        //（即此时的新开始结点）。
    }
}
```

7. 分析：

此题思路不难，可以这样解决，用指针 p 从头至尾扫描 A 链表，当发现结点 data 域为偶数的结点则取下，插入链表 B 中。要用头插法还是用尾插法呢，因为题目要求保持原来数据元素的相对顺序，所以要用尾插法来建立 B 链表。

本题代码如下：

```
void split2(LNode *&A,LNode *&B)
{
    LNode *p,*q,*r;
   B=(LNode*)malloc(sizeof(LNode));  //申请链表 B 的头结点
  B->next=NULL;//每申请一个新结点的时候，将其指针域 next 设置为 NULL 是个好习
                //惯,这样可以避免很多因链表的终端结点忘记置 NULL 而产生的错误。
    r=B;
    p=A;
 while(p->next!=NULL)//p 始终指向当前被判断结点的前驱结点,这和删除结点类似,
 {                   //因为取下一个结点，就是删除一个结点，只是不释放这个
                     //结点的内存空间而已。
    if(p->next->data%2==0) //判断结点的 data 域是否为偶数，是则从链表中取下
    {
```

```
            q=p->next;          //q 指向要从链表中取下的结点
            p->next=q->next;    //从链表中取下这个结点
            q->next=NULL;
            r->next=q;
            r=q;
        }
        p=p->next;          //p 后移一个位置，即开始检查下一个结点。
    }
}
```

（2）思考题

1.分析：

通常在顺序表中找最小值，需要一个循环变量 i 用来控制循环和一个始终记录当前所扫描序列中最小值的变量 min。本题则不同，题目要求只能用 A[0]~A[N-1]和 i 这 N+1 个变量，且要求不能破坏数组 A[]中的数据，这就是说现在我们要用 i 这一个变量来实现通常题目中 i 和 min 两者实现的功能。本题一种可行的办法如下：

i 是 int 型变量，对于处理 N 规模的数据足够用，我们可以让 i 的十位上的数字作为循环变量，将 i 个位上的数字来代替通常题目中 min 的功能，这样一个 i 就可以实现 i 与 min 两者实现的功能。对于本题中的 i，i%10 即取 i 个位上的数字，i/10 即取 i 十位上的数字。

由以上分析可写出如下代码：

```
void FindMin(int A[],int &i)//用 i 来保存最小值
{
    i=A[0];                     //i 先保存存入 A[0]的值。
    while(i/10<=N-1)      //取 i 的十位上的数字作为循环变量,来与 N-1 比较。
    {
        if(i%10>A[i/10]) //取 i 个位上的数字与 A[i/10]中各数值比较。
        {
            i=i-i%10;     //如果过 i 个位上的数字大于 A[i/10]中数字则将 i
            i=i+A[i/10]; //个位上的数字换成 A[i/10]。
        }
        i=i+10;         //i 十位上的数字加 1,即对 A[]中下一个数字进行检测。
    }
    i=i%10;             //循环结束后 i 个位上的数字保存了 A[]中最小值,将 i
                        //更新为 i 个位上的数字。
}
```

2.分析：

本题可用递归的方法解决,在表不空的情况下先递归的逆序打印表中第一个数据之后的数据，然后打印第一个数据，即可实现单链表的中数据的逆序打印。

代码如下：

```
void reprint(LNode *L)
{
    if(L!=NULL)
    {
        reprint(L->next);//递归逆序打印开始结点后边的数据
        cout<<L->data<<" ";//打印开始结点中数据。
    }
}
```