数据结构实验指导书

邢振祥

电子与信息工程系计算机应用教研室
2010-11-18

《数据结构》是计算机科学中一门综合性的专业核心基础课程。它是编译原理、操作系统、数据库系统原理、算法设计与分析及计算机应用方面的主要基础。本课程教学目的是通过本课程的学习，要求学生掌握数据结构的特点、存储方法和基本运算，培养学生运用 C（C++）语言正确编程及调试的能力，运用数据结构解决简单的实际问题的能力，为后续计算机专业课程的学习打下坚实的基础。

本课程总学时为 64 学时，其中理论课程学时为 48 学时，实验学时为 16 学时，实验内容安排如下：

1．线性表的应用（4 学时）

2.栈和队列的应用（2 学时）

3.数组的应用（2 学时）（选作）

4.树和森林的应用（2 学时）

5.图的应用（4 学时）

6．查找方法的实现（2 学时）

7．排序方法的实现（2 学时）

第 1 章 绪 论

本章讨论的是数据结构和算法的基本概念，为巩固理论知识的学习，本章的实验内容针对最基本的数据结构——数组，以及基于数组的简单算法，实现程序设计语言和数据结构的自然衔接，从数据结构的角度重新思考如何进行程序设计，从而提升程序设计乃至算法设计的能力。

1.1 实验的一般步骤

1.1.1 概述

数据结构是一门实践性很强的课程，只靠读书和做习题是不能提高实践能力的，尤其是在数据结构中要解决的问题更接近于实际。数据结构的实验是对学生的一种全面的综合训练，与程序设计语言课程中的实验不同，数据结构课程中的实验多属创造性的活动，需要学生自己进行问题分析、进行数据结构和算法的设计、再上机调试和测试程序。数据结构的实验是一种自主性很强的学习过程，其教学目的主要有两个：⑴ 深化理解和掌握书本上的理论知识，将书本上的知识变“活”；⑵ 理论和实践相结合，使学生学会如何把书本上有关数据结构和算法的知识用于解决实际问题，培养数据结构的应用能力和软件工程所需要的实践能力。

为了达到上述目的，本书安排了如下三类实验：

⑴ 验证实验：其主要内容是将书上的重要数据结构上机实现，深化理解和掌握理论知识，这部分的实验不需要学生自己设计，只需将给定的方案实现即可；

⑵ 设计实验：其主要内容是针对具体问题，应用某一个知识点，自己设计方案，并上机实现，目的是培养学生对数据结构的简单应用能力；

⑶ 综合实验：其主要内容是针对具体问题，应用某几个知识点，自己设计方案，并上机实现，目的是培养学生对数据结构的综合应用能力。

在验证实验中，由实验目的、实验内容、实现提示和实验程序等四部分组成，其中实验目的明确了该实验要学生掌握哪些知识点；实验内容规定了实验的具体任务；实现提示给出了数据结构和算法的设计方法；实验程序给出了实验的范例程序，并且在主教材的随书光盘中有该实验涉及到的数据结构的全部实现。在验证实验中，不要求但鼓励学生在完成实验任务的基础上，对该实验涉及的数据结构的其他实现方法进行探索。

在设计实验和综合实验中，由问题描述、基本要求、设计思想、思考题等四部分组成，其中问题描述是为学生建立问题的背景环境，指明“问题是什么”；基本要求是对问题的实现进行基本规范，保证预定的训练意图，使某些难点和重点不会被绕过去，而且也便于教学检查；设计思想给出了设计数据结构和算法的主要思路；思考题引导学生在做完实验后进行总结和扩充。

虽然在设计实验和综合实验中都给出了一定的设计方案，但是，学生不应拘泥于这些分析和设计，要尽量发挥想象力和创造力。对于一个实际问题，每个人可能会有不同的解决办法，本书给出的范例方案，只是希望把学生的思路引入正轨，提出了思考问题的方法，但是不希望限制学生的思维，鼓励学生自己设计解决方案。

1.1.2 验证实验的一般步骤

验证实验安排的内容在书上都能找到具体的实现方法，并且在主教材的随书光盘中也都有相应的程序实现。这些验证实验是学生在学习完一种数据结构后进行的，对于深化理解和掌握相应数据结构具有很重要的意义。

1. 预备知识的学习

由于篇幅所限，本书没有整理验证实验所用到的预备知识，但主教材中的相关内容已经叙述得很清楚了，需要学生在实验前复习实验所用的预备知识。这需要学生有自主的学习意识和整理知识的能力。

2. 上机前的准备

将实现提示中给出的数据类型和算法转换为对应的程序，并进行静态检查，尽量减少语法错误和逻辑错误。

很多学生在上机时只带一本数据结构书或实验指导书，而书上只有算法设计而没有实验程序，于是就直接在键盘上输入程序，结果不仅程序的输入速度慢，而且编译后出现很多错误。上机前的充分准备能高效利用机时，在有限的时间内完成更多的实验内容。

3. 上机调试和测试程序

调试程序是一个辛苦但充满乐趣的过程，也是培养程序员素质的一个重要环节。很多学生都有这样的经历：化了好长时间去调试程序，错误却越改越多。究其原因，一方面，是对调试工具不熟悉，出现了错误提示却不知道这种错误是如何产生的；另一方面，没有认识到努力预先避免错误的重要性，也就是对程序进行静态检查。

对程序进行测试，首先需要设计测试数据。在数据结构中测试数据需要考虑两种情况：⑴ 一般情况：例如循环的中间数据、随机产生的数据等；⑵ 特殊情况：例如循环的边界条件、数据结构的边界条件等。

4. 实验报告

在实验后要总结和整理实验报告，实验报告的一般格式请参见附录一。

1.1.3 设计实验和综合实验的一般步骤

设计实验和综合实验的自主性比较强，涉及到的知识点也比较多，可以在课程设计中完成，设计实验推荐单人完成，综合实验推荐多人完成，主要目的是为了培养数据结构的应用能力、软件工程的规范训练、团队精神和良好的科学作风。

1. 问题分析

在设计实验和综合实验中的问题描述通常都很简洁，因此，首先要充分理解问题，明确问题要求做什么，限制条件是什么，也就是对所需完成的任务做出明确的描述。例如：输入数据的类型、值的范围以及输入的形式；输出数据的类型、值的范围以及输入的形式；哪些属于非法输入，等等。在问题分析时还应该准备好测试数据。

2. 概要设计

概要设计是对问题描述中涉及到的数据定义抽象数据类型，设计数据结构，设计算法的伪代码描述。在这个过程中，要综合考虑系统的功能，使得系统结构清晰、合理、简单，抽象数据类型尽可能做到数据封闭，基本操作的说明尽可能明确。而不必过早地考虑存储结构，不必过早地考虑语言的实现细节，不必过早地表述辅助数据结构和局部变量。

3. 详细设计

在详细设计阶段，需要设计具体的存储结构（即用 C++描述抽象数据类型对应的类）以及算法所需的辅助数据结构，算法在伪代码的基础上要考虑细节问题并用 C++描述。

此外，还要设计一定的用户界面。数据结构课程实验的主要目的是为了培养数据结构的应用能力，因此在实验中不要求图形界面，只需要在屏幕上提示用户每一步操作的输入、将结果输出即可。

4. 编码实现和静态检查

将详细设计阶段的结果进一步优化为 C++程序，并做静态检查。

很多初学者在编写程序后都有这样的心态：确信自己的程序是正确的，认为上机前的任务已经完成，检查错误是计算机的事。这种心态是极为有害的，这样的上机调试效率是极低的。事实上，即使有几十年经验的高级软件工程师，也经常利用静态检查来查找程序中的错误。

5. 上机调试和测试程序

掌握调试工具，设计测试数据，上机调试和测试程序。调试正确后，认真整理源程序和注释，给出带有完整注释且格式良好的源程序清单和结果。

6. 总结并整理实验报告

在实验后要总结和整理课程设计报告，课程设计报告的一般格式请参见附录二。

1.2 设计实验

1.2.1 在数组中求最小值

1. 问题描述

已知一个数组，求该数组中值最小的元素。

2. 基本要求

⑴ 对于由确定元素组成的数组（即在程序中直接赋值），实现求最小值；

⑵ 随机生成数组元素（即由机器生成随机数），实现求最小值；

⑶ 分析算法的时间性能。

3. 设计思想

我们都知道“打擂台”这个名词，它的意思是说，如果有若干人比武，先有一个人站在台上，再上去一个人与其交手，败者下台，胜者留在台上。如此下去，直到所有人都上台比过为止，最后留在台上的就是胜者。按照这个思路，首先把数组 a[0]的值赋给变量 min，min 就是开始时的擂主，然后让下一个元素与它比较，将二者中值较小者保存在 min 中，直到数组中所有元素都比完为止。最后 min 中保存的就是数组中所有元素的最小值。

4. 算法描述

下面给出具体的在以个整型数组中求最小值的算法。

在数组中求最小值算法

```
int ArrayMin(int a[ ], int n)
{
    min=a[0];
    for (i=1; i<n; i++)
        if (a[i]<min) min=a[i];
    return min;
}
```

【思考题】(1) 在数组中求最大值和最小值，要求算法有较好的时间性能；

(2) 在数组中求最大值和次最大值，要求算法有较好的时间性能。

1.2.2 统计候选人得票

1. 问题描述

设有 *n* 个候选人参加选举，统计每个候选人最后的得票情况。

2. 基本要求

(1) 以数组作为存储结构；

(2) 设计统计得票算法，将最后的得票情况统计并输出。

3. 设计思想

可以将每个候选人设计为一个结构类型，包括名字和得票数，将 *n* 个候选人组成一个结构数组，其存储结构定义如下：

```
const int n=10;     //假设有 10 个人参加选举
struct Person
{
   char *name;
   int count;
} Leader[n];
```

可以从键盘依次输入选举情况，每次输入一个人的名字，将输入的名字与结构数组 Leader 进行比较，将对应候选人的得票数加 1。

4. 算法描述

选票统计算法

```
void Election(Person Leader[ ], int n)
{
   cin>>name;
   while (name!="#")
   {
      for (i=0; i<n; i++)
          if (strcmp(Leader[i].name, name) ==0)   Leader[i].count++;
      cin>>name;
   }
   for (i=0; i<n; i++)
      cout<<Leader[i].name<<"得票数为："<<Leader[i].count<<endl;
}
```

【思考题】将该问题用 C++中的类实现。

1.3 综合实验

1.3.1 顺序查找最好、最坏和平均的时间性能

1. 问题描述

在一维整型数组 A[n]中顺序查找与给定值相等的元素。

2. 基本要求

⑴ 只考虑查找成功的情况；

⑵ 求出最好、最坏、平均情况下待查值与数组元素的实际比较次数；

⑶ 总结实验结果，给出结论。

3. 设计思想

所谓顺序查找就是从数组的第一个元素开始，依次比较每一个元素与待查值是否相等。为了测算待查值与数组元素的实际比较次数，需要在算法中设置计算比较次数的累加器 count，算法结束后输出其值。

4. 算法描述

顺序查找算法

```
int Find(int A[ ], int n, int k)
{
   count=0;
   for (i=0; i<n; i++)
         if (++count  &&  A[i]= =k) break;
   cout<<"比较次数为"<<count<<endl;
   return i;
}
```

【思考题】⑴ 如果考虑查找失败的情况，应如何修改算法？

⑵ 对于排序问题设计一个算法，并分析最好、最坏和平均的时间性能。

第 2 章　线性表

线性表是最简单、最常用的基本数据结构，在实际问题中有着广泛的应用。通过本章的验证实验，巩固对线性表逻辑结构的理解，掌握线性表的存储结构及基本操作的实现，为应用线性表解决实际问题奠定良好的基础。通过本章的设计实验和综合实验，进一步培养以线性表作为数据结构解决实际问题的应用能力。

2.1 验证实验

2.1.1 顺序表操作验证

1. 实验目的

⑴ 掌握线性表的顺序存储结构；

⑵ 验证顺序表及其基本操作的实现；

⑶ 掌握数据结构及算法的程序实现的基本方法。

2. 实验内容

⑴ 建立含有若干个元素的顺序表；

⑵ 对已建立的顺序表实现插入、删除、查找等基本操作。

3. 实现提示

首先定义顺序表的数据类型——顺序表类 SeqList，包括题目要求的插入、删除、查找等基本操作，为便于查看操作结果，设计一个输出函数依次输出顺序表的元素。

```
const int MaxSize=10;
template <class T>          //定义模板类 SeqList
class SeqList
{
public:
    SeqList( ){length=0;}          //无参构造函数
    SeqList(T a[ ], int n);        //有参构造函数
    void Insert(int i, T x);   //在线性表中第 i 个位置插入值为 x 的元素
    T Delete(int i);           //删除线性表的第 i 个元素
    int Locate(T x );          //按值查找，求线性表中值为 x 的元素序号
    void PrintList( );         //遍历线性表，按序号依次输出各元素
private:
    T data[MaxSize];           //存放数据元素的数组
    int length;                //线性表的长度
};
```

其次，建立含有 n 个数据元素的顺序表，即设计构造函数。算法如下：

顺序表有参构造函数 SeqList

```
template <class T>
SeqList:: SeqList(T a[ ], int n)
{
    if (n>MaxSize) throw "参数非法";
    for (i=0; i<n; i++)
      data[i]=a[i];
    length=n;
}
```

最后，对建立的顺序表设计插入、删除、查找等基本操作的算法。

⑴ 插入算法

顺序表插入算法 Insert

```
template <class T>
void SeqList::Insert(int i, T x)
{
  if (length>=MaxSize) throw "上溢";
  if (i<1 || i>length+1) throw "位置";
  for (j=length; j>=i; j--)
    data[j]=data[j-1];     //注意第 j 个元素存在数组下标为 j-1 处
  data[i-1]=x;
  length++;
}
```

⑵ 删除算法

顺序表删除算法 Delete

```
template <class T>
T SeqList::Delete(int i)
{
  if (length==0) throw "下溢";
  if (i<1 || i>length) throw "位置";
  x=data[i-1];
  for (j=i; j<length; j++)
    data[j-1]=data[j];     //注意此处 j 已经是元素所在的数组下标
  length--;
  return x;
}
```

⑶ 查找算法

顺序表按值查找算法 Locate

```
template <class T>
int SeqList::Locate(T x)
{
  for (i=0; i<length; i++)
    if (data[i]==x) return i+1;   //下标为 i 的元素等于 x，返回其序号 i+1
  return 0;                  //退出循环，说明查找失败
}
```

4. 实验程序

```
// 以下为头函数，文件名为 SeqList.h
#ifndef SeqList_H
#define SeqList_H
const int MaxSize=100;   //100 只是示例性的数据，可以根据实际问题具体定义
template <class T>          //定义模板类 SeqList
class SeqList
{
public:
    SeqList( ){length=0;}            //无参构造函数，创建一个空表
    SeqList(T a[ ], int n);          //有参构造函数
    void Insert(int i, T x);     //在线性表中第 i 个位置插入值为 x 的元素
    T Delete(int i);             //删除线性表的第 i 个元素
    int Locate(T x);            //按值查找，求线性表中值为 x 的元素序号
    void PrintList( );           //遍历线性表，按序号依次输出各元素
private:
    T data[MaxSize];          //存放数据元素的数组
    int length;                  //线性表的长度
};
#endif

// 以下为 SeqList 类中成员函数的定义部分，文件名为 SeqList.cpp
#include "SeqList.h"
template <class T>
SeqList<T>:: SeqList(T a[ ], int n)
{
  if (n>MaxSize) throw "参数非法";
  for (int i=0; i<n; i++)
      data[i]=a[i];
  length=n;
}

template <class T>
void SeqList<T>::Insert(int i, T x)
{
  if (length>=MaxSize) throw "上溢";
```

```
    if (i<1 || i>length+1) throw "位置";
    for (int j=length; j>=i; j--)
        data[j]=data[j-1];     //注意第 j 个元素存在数组下标为 j-1 处
    data[i-1]=x;
    length++;
}

template <class T>
T SeqList<T>::Delete(int i)
{
    if (length==0) throw "下溢";
    if (i<1 | | i>length) throw "位置";
    T x=data[i-1];
    for (int j=i; j<length; j++)
        data[j-1]=data[j];     //注意此处 j 已经是元素所在的数组下标
    length--;
    return x;
}

template <class T>
int SeqList<T>::Locate(T x)
{
    for (int i=0; i<length; i++)
        if (data[i]==x)    return i+1 ;   //下标为 i 的元素等于 x，返回其序号 i+1
    return 0;   //退出循环，说明查找失败
}

template <class T>
void SeqList<T>::PrintList( )
{
    for (int i=0; i<length; i++)
        cout<<data[i]<<endl;
}

// 以下为主函数
```

```
#include<iostream.h>            //引用输入输出流库函数的头文件
#include"SeqList.cpp"           //引用顺序表的类声明和定义
void main( )
{
   int r[ ]={1, 2, 3, 4, 5};
   SeqList<int> a(r, 5);
   cout<<"执行插入操作前数据为："<<endl;
   a.PrintList( );                      //输出所有元素
   try
   {
      a.Insert(2,3);
   }
   catch (char *s)
   {
      cout<<s<<endl;
   }
   cout<<"执行插入操作后数据为："<<endl;
   a.PrintList( );                      //输出所有元素
   cout<<"值为 3 的元素位置为:";
   cout<<a.Locate(3)<<endl;             //查找元素 3，并返回在单链表中位置
   cout<<"执行删除第一个元素操作，删除前数据为："<<endl;
   a.PrintList( );                      //输出所有元素
   try
   {
       a.Delete(1);                     //删除元素 1
   }
   catch (char *s)
   {
      cout<<s<<endl;
   }
   cout<<"删除后数据为："<<endl;
   a.PrintList( );                      //输出所有元素
}
```

2.1.2 单链表操作验证

1. 实验目的

⑴ 掌握线性表的链接存储结构；

⑵ 验证单链表及其基本操作的实现；

⑶ 进一步掌握数据结构及算法的程序实现的基本方法。

2. 实验内容

⑴ 用头插法（或尾插法）建立带头结点的单链表；

⑵ 对已建立的单链表实现插入、删除、查找等基本操作。

3. 实现提示

首先，将单链表中的结点定义为如下结构类型：

```
template <class T>
struct Node
{
    T data;
    Node<T> *next;
};
```

其次，定义单链表的数据类型——单链表类 LinkList，包括题目要求的插入、删除、查找等基本操作，为便于查看操作结果，设计一个输出函数依次输出单链表的元素。

```
template <class T>
class LinkList
{
public:
LinkList(T a[ ], int n);   //建立有 n 个元素的单链表
~LinkList( );              //析构函数
void Insert(int i, T x);   //在单链表中第 i 个位置插入元素值为 x 的结点
T Delete(int i);           //在单链表中删除第 i 个结点
int Locate(T x);           //求单链表中值为 x 的元素序号
void PrintList( );         //遍历单链表，按序号依次输出各元素
private:
```

```
Node<T> *first;   //单链表的头指针

};
```

再次，设计单链表类 LinkList 的构造函数和析构函数。

⑴ 用头插法或尾插法建立单链表。头插法建立单链表的算法如下：

头插法建立单链表

```
template <class T>
LinkList:: LinkList(T a[ ], int n)
{
    first=new Node<T>;
    first->next=NULL;   //初始化一个空链表
    for (i=0; i<n; i++)
    {
        s=new Node<T>; s->data=a[i];   //为每个数组元素建立一个结点
        s->next=first->next;           //插入到头结点之后
        first->next=s;
    }
}
```

⑵ 析构函数用于释放单链表中所有结点，算法如下：

单链表的析构函数算法~LinkList

```
template <class T>
LinkList:: ~LinkList( )
{
   p=first;       //工作指针 p 初始化
   while (p)      //释放单链表的每一个结点的存储空间
   {
      q=p;       //暂存被释放结点
      p=p->next;   //工作指针 p 指向被释放结点的下一个结点，使单链表不断开
      delete q;
   }
}
```

最后，对所建立的单链表设计插入、删除、查找等基本操作的算法。

⑴ 插入算法

单链表插入算法 Insert

```
template <class T>
void LinkList::Insert(int i, T x)
{
   p=first; j=0;       //工作指针 p 初始化
   while (p && j<i-1)
   {
       p=p->next;     //工作指针 p 后移
       j++;
   }
   if (!p) throw "位置";
   else {
        s=new Node<T>; s->data=x;   //向内存申请一个结点 s，其数据域为 x
        s->next=p->next;            //将结点 s 插入到结点 p 之后
        p->next=s;
   }
}
```

⑵ 删除算法

单链表的删除算法 Delete

```
template <class T>
T LinkList::Delete(int i)
{
   p=first ; j=0;   //工作指针 p 初始化
   while（p && j<i-1）  //查找第 i-1 个结点
   {
      p=p->next;
      j++;
   }
   if（!p || !p->next）throw "位置";  //结点 p 不存在或结点 p 的后继结点不存在
      else {
            q=p->next; x=q->data;   //暂存被删结点
            p->next=q->next;   //摘链
            delete q;
            return x;
   }
}
```

⑶ 查找算法

单链表查找算法 Locate

```
template <class T>
int LinkList:: Locate(T x)
{
   p=first->next;   j=1;
   while（p && p->data!=x）
   {
      p=p->next;          //工作指针 p 后移
      j++;
   }
   if（p）return j;
      else return 0;
}
```

4. 实验程序

```
// 以下为头函数，文件名为 LinkList.h
#ifndef LinkList_H
#define LinkList_H
template <class T>
struct Node
{
    T data;
    Node<T> *next;   //此处<T>也可以省略
};
```

```
template <class T>
class LinkList
{
   public:
      LinkList(T a[ ], int n);     //建立有 n 个元素的单链表
      ~LinkList( );                //析构函数
      void Insert(int i, T x);     //在单链表中第 i 个位置插入元素值为 x 的结点
      T Delete(int i);             //在单链表中删除第 i 个结点
      int Locate(T x);             //求单链表中值为 x 的元素序号
      void PrintList();                 //遍历单链表，按序号依次输出各元素
  private:
     Node<T> *first;   //单链表的头指针
};
#endif

// 以下为头函数 LinkList.h 中 LinkList 类的成员函数的定义，文件名为 LinkList.cpp
#include"LinkList.h"
template <class T>
LinkList<T>:: LinkList(T a[ ], int n)
{
   first=new Node<T>;
   first->next=NULL;   //初始化一个空链表
   for (int i=0; i<n; i++)
   {
      Node<T> *s;
      s=new Node<T>; s->data=a[i];   //为每个数组元素建立一个结点
      s->next=first->next;           //插入到头结点之后
      first->next=s;
   }
 }

template <class T>
LinkList<T>:: ~LinkList( )
{
```

```
    Node<T> *p, *q;
    p=first;        //工作指针 p 初始化
    while (p)     //释放单链表的每一个结点的存储空间
    {
      q=p;       //暂存被释放结点
      p=p->next;   //工作指针 p 指向被释放结点的下一个结点，使单链表不断开
      delete q;
    }
}

template <class T>
void LinkList<T>::Insert(int i, T x)
{
    Node<T> *p; int j;
    p=first ; j=0;       //工作指针 p 初始化
    while (p && j<i-1)
    {
       p=p->next;     //工作指针 p 后移
       j++;
    }
    if (!p) throw "位置";
    else {
        Node<T> *s;
        s=new Node<T>;
        s->data=x;   //向内存申请一个结点 s，其数据域为 x
        s->next=p->next;            //将结点 s 插入到结点 p 之后
        p->next=s;
    }
}

template <class T>
T LinkList<T>::Delete(int i)
{
   Node<T> *p; int j;
   p=first ; j=0;   //工作指针 p 初始化
```

```
    while (p && j<i-1)   //查找第 i-1 个结点
    {
        p=p->next;
        j++;
    }
    if (!p || !p->next) throw "位置";   //结点 p 不存在或结点 p 的后继结点不存在
        else {
                Node<T> *q; int x;
                q=p->next; x=q->data;   //暂存被删结点
                p->next=q->next;   //摘链
                delete q;
                return x;
        }
}

template <class T>
int LinkList<T>::Locate(T x)
{
        Node<T> *p; int j;
        p=first->next; j=1;
        while (p && p->data!=x)
        {
            p=p->next;
            j++;
        }
        if (p) return j;
        else return 0;
}

template <class T>
void LinkList<T>::PrintList( )
{
        Node<T> *p;
        p=first->next;
        while (p)
```

```
    {
      cout<<p->data<<endl;
      p=p->next;
    }
}

// 以下为主函数
#include<iostream.h>     //引用输入输出流库函数的头文件
#include"LinkList.cpp"   //引用单链表类的声明和定义
void main( )
{
  int r[ ]={1, 2, 3, 4, 5};
  LinkList<int> a(r, 5);
  cout<<"执行插入操作前数据为："<<endl;
  a.PrintList( );                    //显示链表中所有元素
  try
  {
    a.Insert(2, 5);
  }
  catch (char *s)
  {
    cout<<s<<endl;
  }
  cout<<"执行插入操作后数据为："<<endl;
  a.PrintList( );                    //显示链表中所有元素
  cout<<"值为 5 的元素位置为:";
  cout<<a.Locate(5)<<endl;           //查找元素 5，并返回在单链表中位置
  cout<<"执行删除操作前数据为："<<endl;
  a.PrintList( );                    //显示链表中所有元素
  try
  {
    a.Delete(1);                     //删除元素 4
  }
  catch (char *s)
  {
```

```
        cout<<s<<endl;
    }
    cout<<"执行删除操作后数据为："<<endl;
    a.PrintList( );                    //显示链表中所有元素
}
```

【思考题】为单链表的结点设计一个结点类，重新实现单链表基本操作的验证。

2.2 设计实验

2.2.1 数组的循环移位

1. 问题描述

对于一个给定的整型数组循环右移 *i* 位。

2. 基本要求

⑴ 在原数组中实现循环右移，不另外申请空间；

⑵ 时间性能尽可能好；

⑶ 分析算法的时间复杂度。

3. 设计思想

将这个问题看作是把数组 ab 转换成数组 ba（a 代表数组的前 *i* 个元素，b 代表数组中余下的 *n-i* 个元素），先将 a 逆置得到 a^rb，再将 b 逆置得到 a^rb^r，最后将整个 a^rb^r 逆置得到$(a^rb^r)^r$=ba。设 Reverse 函数执行将数组元素逆置的操作，对 abcdefgh 向左循环移动 3 个位置的过程如下：

```
Reverse(0, i-1);     //得到 cbadefgh
Reverse(i, n-1);     //得到 cbahgfed
Reverse(0, n-1);     //得到 defghabc
```

具体分析过程请参见主教材第一章思想火花。

4. 算法描述

循环右移算法

```
void Converse(int A[ ], int n, int i)
{
   Reverse(A, 0, i-1);     //前 i 个元素逆置
   Reverse(A, i, n-1);     //后 n-i 个元素逆置
   Reverse(A, 0, n-1);     //整个数组逆置
}
void Reverse(int A[ ], int from, int to)       //将数组 A 中元素从 from 到 to 逆置
{
   for (i=0; i<(to-from+1)/2; i++)
        A[from+i]←→A[to-i];       //交换元素
}
```

【思考题】你还有其他解决方法吗？请设计算法并上机实现。

2.2.2 集合的交、并和差运算的实现

1. 问题描述

用有序单链表表示集合，实现集合的交、并和差运算。

2. 基本要求

⑴ 对集合中的元素，用有序单链表进行存储；

⑵ 实现交、并、差运算时，不另外申请存储空间；

⑶ 充分利用单链表的有序性，算法有较好的时间性能。

3. 设计思想

首先，建立两个带头结点的有序单链表表示集合 A 和 B。单链表的结点结构和建立算法请参见2.1.2，需要注意的是：利用头插法建立有序单链表，实参数组应该是降序排列。

其次，根据集合的运算规则，利用单链表的有序性，设计交、并和差运算。

⑴ 根据集合的运算规则，集合 $A \cap B$ 中包含所有既属于集合 A 又属于集合 B 的元素。因此，需查找单链表 A 和 B 中的相同元素并保留在单链表 A 中。算法如下：

求集合的交集算法

```
void Interest(Node *A, Node *B)
{   //A、B 分别是两个单链表的头指针，最后的结果在单链表 A 中
    pre=A; p=A->next; q=B->next;
    while (p && q)
    {
        if (p->data<q->data) {
                                pre->next=p->next;
                                p=pre->next;
        }
        else if (p->data>q->data) q=q->next;
             else {
                    p=p->next;
                    q=q->next;
                  }
    }
}
```

⑵ 根据集合的运算规则，集合 $A \cup B$ 中包含所有或属于集合 A 或属于集合 B 的元素。因此，对单链表 B 中的每个元素 x，在单链表 A 中进行查找，若存在和 x 不相同的元素，则将该结点插入到单链表 A 中。算法请参照求集合的交集自行设计。

⑶ 根据集合的运算规则，集合 A-B 中包含所有属于集合 A 而不属于集合 B 的元素。因此，对单链表 B 中的每个元素 x，在单链表 A 中进行查找，若存在和 x 相同的结点，则将该结点从单链表 A 中删除。算法请参照求集合的交集自行设计。

【思考题】如果表示集合的单链表是无序的，应如何实现集合的交、并和差运算？

2.3 综合实验

2.3.1 约瑟夫环问题

1. 问题描述

设有编号为 1，2，……，*n* 的 *n*(*n*>0)个人围成一个圈，每个人持有一个密码 *m*，从第 1 个人开始报数，报到 *m* 时停止报数，报 *m* 的人出圈，再从他的下一个人起重新报数，报到 *m* 时停止报数，报 *m* 的出圈，……，如此下去，直到所有人全部出圈为止。当任意给定 *n* 和 *m* 后，设计算法求 *n* 个人出圈的次序。

2. 基本要求

⑴ 建立模型，确定存储结构；

⑵ 对任意 *n* 个人，密码为 *m*，实现约瑟夫环问题；

⑶ 出圈的顺序可以依次输出，也可以用一个数组存储。

3. 设计思想

首先，设计实现约瑟夫环问题的存储结构。由于约瑟夫环问题本身具有循环性质，考虑采用循环链表，为了统一对表中任意结点的操作，循环链表不带头结点。将循环链表的结点定义为如下结构类型：

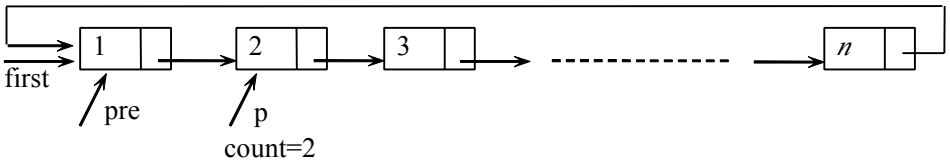
```
struct Node
{
   int data;   //编号
   Node *next;
};
```

其次，建立一个不带头结点的循环链表并由头指针 first 指示。具体算法请参见 2.1.2 自行设计。

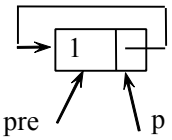
最后，设计约瑟夫环问题的算法。下面给出伪代码描述，操作示意图如图 2-1 所示。

伪代码

1. 工作指针 pre 和 p 初始化，计数器 count 初始化；
   pre=first; p=first->next; count=2; //为便于删除操作，从 2 开始计数
2. 循环直到 p=pre
   2.1 如果 count=m，则
      2.1.1 输出结点 p；
      2.1.2 删除结点 p；
      2.1.3 计数器 count 清零，重新开始计数；
   2.2 否则，执行
      2.2.1 工作指针 pre 和 p 后移；
      2.2.2 计数器增 1；
3. 退出循环，链表中只剩下一个结点 p，输出结点 p 后将结点 p 删除；



(a) 一般情况



(b) 只剩下一个结点的特殊情况

图 2-1　约瑟夫环问题存储示意图

【思考题】⑴ 采用顺序存储结构如何实现约瑟夫环问题？

⑵ 如果每个人持有的密码不同，应如何实现约瑟夫环问题？

第 3 章 栈、队列和串

本章的实验内容围绕三种特殊线性表——栈、队列和串展开。

栈和队列广泛应用在各种软件系统中，掌握栈和队列的存储结构及基本操作的实现是以栈和队列作为数据结构解决实际问题的基础，尤其栈有很多经典应用，深刻理解并实现这些经典应用，对于提高数据结构和算法的应用能力具有很重要的作用。

尽管在大多数高级语言中都提供了串变量并实现了串的基本操作，但在实际应用中，串往往具有不同的特点，要实现串的处理，就必须根据具体情况采用（或设计）合适的存储结构。本章安排的有关串的实验是灵活运用串的基础。

3.1 验证实验

3.1.1 栈操作验证

1. 实验目的

⑴ 掌握栈的顺序存储结构；

⑵ 掌握栈的操作特性；

⑶ 掌握基于顺序栈的基本操作的实现方法。

2. 实验内容

⑴ 建立一个空栈；

⑵ 对已建立的栈进行插入、删除、取栈顶元素等基本操作。

3. 实现提示

首先，定义顺序栈的数据类型——顺序栈类 SeqStack，包括入栈、出栈、取栈顶元素等基本操作。

```
const int StackSize=10;
template <class T>      //定义模板类 SeqStack
class SeqStack
{
public:
    SeqStack( );              //构造函数，初始化一个空栈
    void Push(T x);           //将元素 x 入栈
    T Pop( );                 //将栈顶元素弹出
    T GetTop( );              //取栈顶元素（并不删除）
private:
    T data[StackSize];    //存放栈元素的数组
    int top;              //栈顶指针，指示栈顶元素在数组中的下标
};
```

其次，设计顺序栈类 SeqStack 的构造函数。初始化一个空栈的算法如下：

顺序栈初始化算法

```
template <class T>
void SeqStack:: SeqStack( )
{
   top=-1;
}
```

最后，对建立的栈设计算法完成插入、删除、取栈顶元素等基本操作。

⑴ 入栈算法

顺序栈入栈算法 Push

```
template <class T>
void SeqStack::Push(T x)
{
   if (top== StackSize-1) throw "上溢";
   top++;
   data[top]=x;
}
```

⑵ 出栈算法

顺序栈出栈算法 Pop

```
template <class T>
T SeqStack:: Pop( )
{
   if (top==-1) throw "下溢";
   x=data[top--];
   return x;
}
```

⑶ 取栈顶元素算法

取栈顶元素算法与出栈算法类似，只是注意不要修改栈顶指针 top。

4. 实验程序

参见光盘。

3.1.2 队列操作验证

1. 实验目的

⑴ 掌握队列的链接存储结构；

⑵ 掌握队列的操作特性；

⑶ 掌握基于链队列的基本操作的实现方法。

2. 实验内容

⑴ 建立一个空队列；

⑵ 对已建立的队列进行插入、删除、取队头元素等基本操作。

3. 实现提示

首先，定义链队列的数据类型——链队列类 LinkQueue，包括入队、出队、取队头元素等基本操作。

```
template <class T>
class LinkQueue
{
public:
    LinkQueue( );          //构造函数，初始化一个空的链队列
    ~LinkQueue( );         //析构函数，释放链队列中各结点的存储空间
    void EnQueue(T x);     //将元素 x 入队
    T DeQueue( );          //将队头元素出队
    T GetQueue( );         //取链队列的队头元素
private:
    Node<T> *front, *rear;   //队头和队尾指针，分别指向头结点和终端结点
};
```

链队列的结点结构和单链表的结点结构相同，请参见 2.1.2 相关内容。

其次，设计构造函数和析构函数。

⑴ 构造函数（初始化一个空队列）的算法如下：

链队列构造函数算法 LinkQueue

```
template <class T>
LinkQueue::LinkQueue( )
{
  s=new Node<T>; s->next=NULL;   //创建一个头结点 s
  front=rear=s;                  //将队头指针和队尾指针都指向头结点 s
}
```

⑵ 析构函数的算法请读者仿照单链表类 LinkList 的析构函数自行设计。

最后，设计算法完成队列的入队（插入）、出队（删除）、取队头元素等基本操作。

⑴ 入队算法

链队列入队算法 EnQueue

```
template <class T>
void LinkQueue::EnQueue(T x)
{
  s=new Node<T>; s->data=x;   //申请一个数据域为 x 的结点 s
  s->next=NULL;
  rear->next=s;               //将结点 s 插入到队尾
```

⑵ 出队算法

链队列出队算法 DeQueue

```
template <class T>
T LinkQueue::DeQueue( )
{
   if (rear==front) throw "下溢";
   p=front->next; x=p->data;      //暂存队头元素
   front->next=p->next;           //将队头元素所在结点摘链
   if (p->next==NULL) rear=front;   //判断出队前队列长度是否为 1
   delete p;
   return x;
}
```

⑶ 取队头元素算法

取队头元素算法与出队算法类似，只是不将队头元素删除。

4. 实验程序

参见光盘。

3.2 设计实验

3.2.1 汗诺塔问题

1. 问题描述

汉诺塔问题来自一个古老的传说：在世界刚被创建的时候有一座钻石宝塔（塔 A），其上有 64 个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔（塔 B 和塔 C）。从世界创始之日起，婆罗门的牧师们就一直在试图把塔 A 上的碟子移动到塔 C 上去，其间借助于塔 B 的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。当牧师们完成任务时，世界末日也就到了。

2. 基本要求

⑴ 设计数据结构，表示三座宝塔和n个盘子；

⑵ 输出每一次移动盘子的情况；

⑶ 分析算法的时间性能。

3. 设计思想

对于汉诺塔问题的求解，可以通过以下三个步骤实现：

⑴ 将塔A上的n-1个碟子借助塔C先移到塔B上。

⑵ 把塔A上剩下的一个碟子移到塔C上。

⑶ 将 *n*-1 个碟子从塔 B 借助于塔 A 移到塔 C 上。

显然，这是一个递归求解的过程，当 *n*=3 时的求解过程如图 3-1 所示。

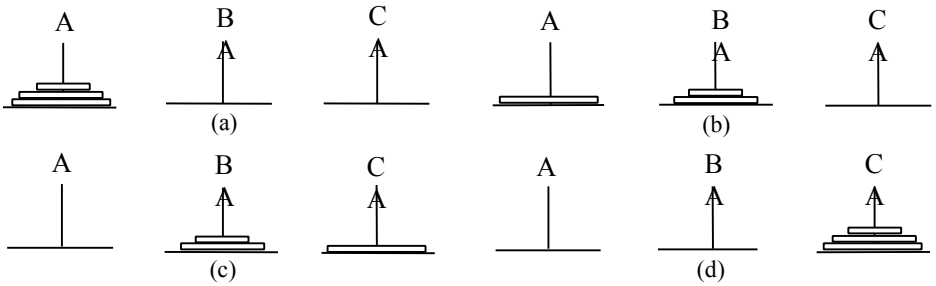


图 3-1　Hanoi 求解示意图

三座宝塔（塔 A、塔 B、塔 C）分别用三个字符 A、B、C 来表示，*n* 个盘子用从 1 开始的连续自然数编号。具体算法如下：

汉 诺 塔 算 法 Hanoi

```
void Hanoi(int n, char A, char B, char C)
{
    if (n==1) cout<<"A→C";     //将碟子从 A 移到 C 上
      else {
            Hanoi(n-1, A, C, B);
            cout<<"A→C";
            Hanoi(n-1, B, A, C);
      }
}
```

【思考题】你能设计一个非递归算法解决汉诺塔问题吗？

3.3 综合实验

3.3.1 表达式求值

1. 问题描述

对一个合法的中缀表达式求值。

假设表达式只包含+、-、×、÷ 四个双目运算符，且运算符本身不具有二义性。

2. 基本要求

⑴ 正确解释表达式；

⑵ 符合四则运算规则；

⑶ 输出最后的计算结果。

3. 设计思想

对中缀表达式求值，通常使用“算符优先算法”。根据四则运算规则，在运算的每一步中，任意两个相继出现的运算符 t 和 c 之间的优先关系至多是下面三种关系之一：

⑴ t<c：t 的优先级低于 c；

⑵ t=c：t 的优先级等于 c；

⑶ t>c：t 的优先级高于 c。

为实现算符优先算法，可以使用两个工作栈：一个栈 OPTR 存放运算符；另一个栈 OPND 存放操作数，中缀表达式用一个字符串数组存储。算法用伪代码描述如下：

伪代码

1. 初始化：栈OPTR中只有一个元素'# '，栈OPND为空栈；
2. 依次读入字符，执行下述操作：
   2.1 ch=读入的字符;
   2.2 若 ch='# '，则算法结束，返回栈 OPND 的栈顶元素；
   2.3 若 ch 不是运算符，则将 ch 入栈 OPND；
   2.4 若 ch 是运算符，则
      2.4.1 t=取栈 OPTR 的栈顶元素；
      2.4.2 比较 ch 和 t 的优先级，执行下述三种操作之一：
         ⑴ 若 ch=t：将栈 OPTR 的栈顶元素出栈；
         ⑵ 若 ch>t：将 ch 插入栈 OPTR 中；
         ⑶ 若 ch<t：在栈 OPND 中弹出两个元素，与 ch 进行运算，将结果插入栈 OPND 中；

【思考题】⑴ 如果要求输出每一步的计算过程，应如何修改算法？

⑵ 如果运算符包含括号且括号可以嵌套，应如何修改算法？

第 4 章 数组和广义表

数组是人们非常熟悉的基本数据结构，科学计算中的矩阵在程序设计语言中就是采用数组实现的。通过本章的验证实验和设计实验，巩固对特殊矩阵和稀疏矩阵的压缩存储方法的理解和运用，在综合实验中安排了两个利用数组实现的简单游戏，从而理解数组在实际问题中的应用。

广义表作为一种特殊的数据结构，兼有线性表、树、图的特性，所以，本章的实验仅实现广义表的基本操作。

4.1 验证实验

4.1.1 对称矩阵的压缩存储

1. 实验目的

⑴ 掌握对称矩阵的压缩存储方法；

⑵ 掌握对称矩阵压缩存储的寻址方法。

2. 实验内容

⑴ 建立一个 $n\times n$ 的对称矩阵；

⑵ 将对称矩阵用一维数组存储；

3. 实现提示

首先建立一个 $n\times n$ 的对称矩阵 A 并初始化矩阵的元素。对称矩阵只需存储下三角部分，即将一个 $n\times n$ 的对称矩阵用一个大小为 $n\times(n+1)/2$ 的一维数组 SA 来存储，则下三角中的元素 a_{ij}（$i\geq j$）在 SA 中的下标 k 与 i、j 的关系为 $k=i\times(i-1)/2+j$。算法如下：

对称矩阵的压缩存储算法

```
void Matrix (int A[ ][ ], int n, int SA[ ])
{
    for (i=0; i<n; i++)
       for (j=0; j<n; j++)
          A[i][j]=i+j;              //生成对称矩阵元素
    for (i=0; i<n; i++)
       for (j=0; j<=i; j++)
          SA[i*(i-1)/2+j]=A[i][j];     //压缩存储
}
```

4. 实验程序

参见光盘。

4.2 设计实验

4.2.1 稀疏矩阵的转置

1. 问题描述

采用三元组顺序表存储稀疏矩阵，并实现其转置。

2. 基本要求

⑴ 设计存储结构实现稀疏矩阵的压缩存储；

⑵ 设计算法实现稀疏矩阵的转置；

⑶ 分析算法的时间复杂度和空间复杂度。

3. 设计思想

将稀疏矩阵的非零元素对应的三元组（行号、列号、非零元素值）所构成的集合，按行优先的顺序排列成一个线性表，对该线性表采用顺序存储结构，同时存储该矩阵的行数、列数和非零元素的个数。三元组顺序表的存储结构定义如下：

```
template <class T>
struct element
{
   int row, col;
   T item
};
const int MaxTerm=10;
struct SparseMatrix
{
   element data[MaxTerm];
   int mu, nu, tu;     //行数、列数、非零元个数
};
```

将稀疏矩阵 A 转置为矩阵 B 的基本思想是：在 A 的三元组顺序表中依次找第 0 列、第 1 列、……、直到最后一列的三元组，并将找到的每个三元组的行、列交换后顺序存储到 B 的三元组顺序表中。算法用伪代码描述如下：

伪代码

```
1. 设置转置后矩阵 B 的行数、列数和非零元素的个数；
2. 在 B 中设置初始存储位置 pb；
3. for（col=最小列号; col<=最大列号; col++）
     3.1 在 A 中查找列号为 col 的三元组；
     3.2 交换其行号和列号，存入 B 中 pb 位置；
     3.3 pb++；
```

【思考题】将主教材中稀疏矩阵转置的第二种方法实现，并与第一种方法进行比较。

4.3 综合实验

4.3.1 魔方阵

1. 问题描述

魔方阵，又叫幻方阵，在我国古代称为“纵横图”。它是在一个 $n \times n$ 的矩阵中填入 1 到 n^2 的数字

（*n* 为奇数），使得每一行、每一列、每条对角线的累加和都相等。例如图 4-2 就是一个 3 阶魔方阵。

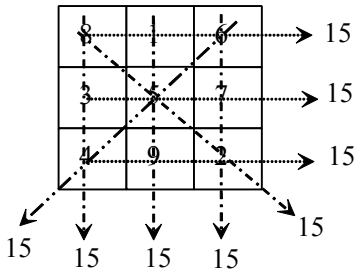


图 4-2　3 阶魔方阵示例

2. 基本要求

⑴ 设计数据结构；

⑵ 设计算法完成任意 n 阶魔方阵的填数；

⑶ 分析算法的时间复杂度。

3. 设计思想

解决魔方阵问题的方法很多，下面介绍一种被称为“左上斜行法”的填数方法，具体填数过程如下：

⑴ 由 1 开始填数，将 1 放在第 0 行的中间位置；

⑵ 将魔方阵想象成上下、左右相接，每次往左上角走一步，会有下列情况：

· 左上角超出上边边界，则在最下边相对应的位置填入下一个数字；

· 左上角超出左边边界，则在最右边相对应的位置填入下一个数字；

· 如果按上述方法找到的位置已填入数据，则在同一列下一行填入下一个数字。

算法如下：

魔方阵算法 Square

```
void Square(int a[ ][ ], int n)
{
   p=0; q=(n-1)/2;
   a[0][q]=1;       //在第 0 行的中间位置填 1
   for (i=2; i<=n*n, i++)
   {
      p=(p-1+n) % n;          //求 i 所在行号
      q=(q-1+n) % n;          //求 i 所在列号
      if (a[p][q]>0) p=(p+1) % n;   //如果位置(p, q)已经有数，填入同一列下一行
      a[p][q]=i;
   }
}
```

【思考题】你还能设计出其他求 *n* 阶魔方阵的方法吗？或者上网查找关于魔方阵的解法。

第 5 章 树和二叉树

树结构是一种非常重要的非线性结构，为实际问题中具有层次关系的数据提供了一种自然的表示方法。本章的实验内容围绕树和二叉树的实现及其实际应用展开，通过本章实验，可以更好地将树结构与实际应用中具有层次结构的问题联系起来，培养在实际问题中应用树结构的能力。

5.1 验证实验

5.1.1 二叉树操作验证

1. 实验目的

⑴ 掌握二叉树的逻辑结构；

⑵ 掌握二叉树的二叉链表存储结构；

⑶ 掌握基于二叉链表存储的二叉树的遍历操作的实现。

2. 实验内容

⑴ 建立一棵含有 *n* 个结点的二叉树，采用二叉链表存储；

⑵ 前序（或中序、后序）遍历该二叉树。

3. 实现提示

二叉链表的结点结构如图 5-2 所示。

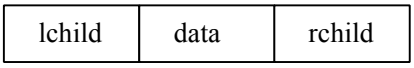


图 5-2　二叉树的结点结构

二叉链表的结点用 C++中的结构类型描述为：

```
template <class T>
struct BiNode
{
   T data;
   BiNode<T> *lchild, *rchild;
};
```

设计实验用二叉链表类 BiTree，类中包含遍历操作。

```
template <class T>
class BiTree
{
public:
   BiTree(BiNode<T> *root); //有参构造函数，初始化一棵二叉树，其前序序列由键盘输入
   ~BiTree( );                    //析构函数，释放二叉链表中各结点的存储空间
```

```
    void PreOrder(BiNode<T> *root);        //前序遍历二叉树
    void InOrder(BiNode<T> *root);         //中序遍历二叉树
    void PostOrder(BiNode<T> *root);       //后序遍历二叉树
private:
    BiNode<T> *root;     //指向根结点的头指针
     void Creat(BiNode<T> *root);         //有参构造函数调用
    void Release(BiNode<T> *root);     //析构函数调用
};
```

设计构造函数，建立一棵二叉树的二叉链表存储。

将二叉树中每个结点的空指针引出一个虚结点，其值为一特定值如“#”，以标识其为空，把这样处理后的二叉树称为原二叉树的扩展二叉树。扩展二叉树的一个遍历序列就能唯一确定一棵二叉树。假设扩展二叉树的前序遍历序列由键盘输入，root 为指向根结点的指针，二叉链表的建立过程是：首先输入根结点，若输入的是一个“#”字符，则表明该二叉树为空树，即 root=NULL；否则输入的字符应该赋给 root->data,，之后依次递归建立它的左子树和右子树。建立二叉链表的递归算法如下：

二叉树的构造函数算法 BiTree

```
template <class T>
BiTree ::BiTree(BiNode<T> *root)
{
   root =creat( );
}

template <class T>
BiNode<T> *BiTree ::Creat( )
{
      cin>>ch;
      if (ch=='# ') return NULL;       //建立一棵空树
       else {
            root=new BiNode<T>;     //生成一个结点
            root->data=ch;
            root->lchild=Creat( );     //递归建立左子树
            root->rchild=Creat( );     //递归建立右子树
      }
}
```

前序遍历的递归算法如下：

二叉树前序遍历递归算法 PreOrder

```
template <class T>
void BiTree::PreOrder(BiNode<T> *root)
{
   if (root ==NULL)    return;        //递归调用的结束条件
      else {
```

中序和后序遍历的算法请仿照前序遍历的递归算法自行设计。

4. 实验程序

参见光盘。

5.2 设计实验

5.2.1 求二叉树中叶子结点的个数

1. 问题描述

已知一棵二叉树，求该二叉树中叶子结点的个数。

2. 基本要求

⑴ 采用二叉链表作存储结构；

⑵ 设计递归算法求叶子结点的个数。

3. 设计思想

求二叉树中叶子结点的个数，即求二叉树的所有结点中左、右子树均为空的结点个数之和。因此可以将此问题转化为遍历问题，在遍历中“访问一个结点”时判断该结点是不是叶子，若是则将计数器累加。算法如下：

求二叉树叶子结点个数算法

```
void CountLeaf(BiNode<T> *root, int &count)
//前序遍历根指针为 root 的二叉树以计算叶子数 count，假定 count 的初值为 0；
{
    if (root!=NULL) {
        if (root->lchild==NULL && root->rchild = =NULL)
            count++;   //若 root 所指的结点是叶子，则计数器加 1;
        CountLeaf(root->lchild, count);     //累计左子树上的叶子数；
        CountLeaf(root->rchild, count);     //累计右子树上的叶子数；
    }
}
```

【思考题】如何设计非递归算法求叶子结点的个数？

5.3 综合实验

5.3.1 哈夫曼编码

1. 问题描述

设某编码系统共有 n 个字符，使用频率分别为$\{w_1, w_2, \ldots, w_n\}$，设计一个不等长的编码方案，使得该编码系统的空间效率最好。

2. 基本要求

⑴ 设计数据结构；

⑵ 设计编码算法；

⑶ 分析时间复杂度和空间复杂度。

3. 设计思想

利用 Huffman 编码树求得最佳的编码方案。

根据哈夫曼算法，建立哈夫曼树时，可以将哈夫曼树定义为一个结构型的一维数组 HuffTree，保存哈夫曼树中各结点的信息，每个结点包括：权值、左孩子、右孩子、双亲，如图 5-4 所示。由于哈夫曼树中共有 $2n$-1 个结点，并且进行 n-1 次合并操作，所以该数组的长度为 $2n$-1。

weight	lchild	rchild	parent

图 5-4　哈夫曼树的结点结构

构造哈夫曼树的伪代码如下：

伪代码

```
1. 数组 huffTree 初始化，所有元素结点的双亲、左右孩子都置为-1；
2. 数组 huffTree 的前 n 个元素的权值置给定权值 w[n]；
3. 进行 n-1 次合并
   3.1 在二叉树集合中选取两个权值最小的根结点，其下标分别为 i1, i2；
   3.2 将二叉树 i1、i2 合并为一棵新的二叉树 k；
```

在哈夫曼树中，设左分支为 0，右分支为 1，从根结点出发，遍历整棵哈夫曼树，求得各个叶子结点所表示字符的哈夫曼编码。

【思考题】对于采用哈夫曼编码树进行的编码，如何设计解码算法？

第 6 章 图

图是最复杂的一种数据结构，同时也是表达能力最强的一种数据结构，其应用十分广泛，很多问题都可以用图来表示。本章的实验内容针对基于邻接矩阵和邻接表存储的图及其基本操作的实现展开，并结合图的具体应用，培养在实际问题中应用图结构的能力。

6.1 验证实验

6.1.1 邻接矩阵操作验证

1. 实验目的

⑴ 掌握图的逻辑结构；

⑵ 掌握图的邻接矩阵存储结构；

⑶ 掌握图在邻接矩阵存储结构上遍历算法的实现。

2. 实验内容

⑴ 建立无向图的邻接矩阵存储；

⑵ 对建立的无向图，进行深度优先遍历；

⑶ 对建立的无向图，进行广度优先遍历。

3. 实现提示

本题采用图的邻接矩阵存储，即用一维数组存储图中顶点的信息，用二维数组存储图中边的信息（即各顶点之间的邻接关系）。

假设无向图 $G=(V, E)$ 有 n 个顶点，则邻接矩阵是一个 $n\times n$ 的方阵，定义为：

$$\text{arc[i][j]}=\begin{cases}1 & \text{若}(v_i, v_j)\in E\\ 0 & \text{否则}\end{cases}$$

设计实验用邻接矩阵类 MGraph，包括遍历操作。

```
const int MaxSize=10;     //图中最多顶点个数
template <class T>
class MGraph
{
public:
    MGraph(T a[ ], int n, int e);     //构造函数，初始化具有 n 个顶点 e 条边的无向图
    void DFSTraverse(int v);          //深度优先遍历图
    void BFSTraverse(int v);          //广度优先遍历图
private:
     T vertex[MaxSize];               //存放图中顶点的数组
     int arc[MaxSize][MaxSize];       //存放图中边的数组
     int vertexNum, arcNum;           //图的顶点数和边数
```

};

⑴ 设计构造函数，建立一个无向图的邻接矩阵存储。

假设图的顶点信息存放在数组 a[n]中，边的信息（即边所依附的顶点）由键盘输入，建立一个无向图的邻接矩阵存储的算法如下：

邻接矩阵构造函数算法 MGraph

```
template <class T>
MGraph::MGraph(T a[ ], int n, int e)
{
    vertexNum=n; arcNum=e;
    for (i=0; i<vertexNum; i++)
        vertex[i]=a[i];
    for (i=0; i<vertexNum; i++)        //初始化邻接矩阵
        for (j=0; j<vertexNum; j++)
            arc[i][j]=0;
    for (k=0; k<arcNum; k++)   //依次输入每一条边，并修改邻接矩阵的相应元素
    {
        cin>>i>>j;          //边依附的两个顶点的序号
        arc[i][j]=1;         //置有边标志
        arc[j][i]=1;
    }
}
```

⑵ 深度优先遍历算法

深度优先遍历算法 DFSTraverse

```
template <class T>
void MGraph::DFSTraverse(int v)
{
    cout<<vertex[v]; visited [v]=1;
    for (j=0; j<vertexNum; j++)
        if (arc[v][j]==1 && visited[j]==0) DFSTraverse( j );
}
```

⑶ 广度优先遍历算法

广度优先遍历算法 BFSTraverse

```
template <class T>
void MGraph::BFSTraverse(int v)
{
    front=rear=-1;     //初始化队列,假设队列采用顺序存储且不会发生溢出
    cout<<vertex[v]; visited[v]=1;    Q[++rear]=v;     //被访问顶点入队
    while (front!=rear)
    {
        v=Q[++front];     //将队头元素出队并送到 v 中
        for (j=0; j<vertexNum; j++)
            if (arc[v][j]==1 && visited[j]==0 ) {
                cout<<vertex[j]; visited[j]=1; Q[++rear]=j;
            }
    }
}
```

4. 实验程序

参见光盘。

6.1.2 邻接表操作验证

1. 实验目的

⑴ 掌握图的逻辑结构；

⑵ 掌握图的邻接表存储结构；

⑶ 掌握图的邻接表存储结构下遍历算法的实现。

2. 实验内容

⑴ 建立一个有向图的邻接表存储结构；

⑵ 对建立的有向图，进行深度优先遍历；

⑶ 对建立的有向图，进行广度优先遍历。

3. 实现提示

邻接表存储结构是基于顺序存储与链接存储相结合的存储方法，基本思想是：对于图的每个顶点 v_i，将所有邻接于 v_i的顶点链成一个单链表，称为顶点 v_i的边表（对于有向图则称为出边表），所有顶点的边表的头指针和存储顶点信息的一维数组构成了顶点表。所以，在邻接表中存在两种结点结构，分别是顶点表结点和边表结点，如图 6-1 所示。

顶点表结点　　边表结点

图 6-1　邻接表表示的结点结构

用 C++中的结构类型描述上述结点。

```
struct ArcNode     //定义边表结点
{
   int adjvex;    //邻接点域
   ArcNode *next;
};
template <class T>
struct VertexNode     //定义顶点表结点
{
   T vertex;
   ArcNode *firstedge;
};
```

设计实验用邻接表类 ALGraph，包括遍历操作。

```
const int MaxSize=10;       //图的最大顶点数
template <class T>
class ALGraph
{
public:
    ALGraph(T a[ ], int n, int e);     //构造函数，初始化一个有 n 个顶点 e 条边的有向图
    ~ALGraph( );       //析构函数，释放邻接表中各边表结点的存储空间
    void DFSTraverse(int v);          //深度优先遍历图
    void BFSTraverse(int v);          //广度优先遍历图
private:
    VertexNode adjlist[MaxSize];     //存放顶点表的数组
    int vertexNum, arcNum;            //图的顶点数和边数
};
```

⑴ 设计构造函数，建立有向图的邻接表存储。

邻接表构造函数算法 ALGraph

```
template <class T>
ALGraph::ALGraph(T a[ ], int n, int e)
{
    vertexNum=n; arcNum=e;
    for (i=0; i<vertexNum; i++)   //输入顶点信息，初始化顶点表
    {
        adjlist[i].vertex=a[i];
        adjlist[i].firstedge=NULL;
    }
    for (k=0; k<arcNum; k++)   //依次输入每一条边，并在相应边表中插入结点
    {
        cin>>i>>j;       //输入边所依附的两个顶点的序号
        s=new ArcNode; s->adjvex=j;   //生成一个边表结点 s
        s->next=adjlist[i].firstedge;       //将结点 s 插入到结点 i 的边表的表头
        adjlist[i].firstedge=s;
    }
}
```

⑵ 深度优先遍历算法如下：

深度优先遍历算法 DFSTraverse

```
template <class T>
void ALGraph::DFSTraverse(int v)
{
    cout<<adjlist[v].vertex;   visited[v]=1;
    p=adjlist[v].firstedge;
    while (p)            //依次搜索顶点 v 的邻接点 j
    {
        j=p->adjvex;
        if (visited[j]==0) DFSTraverse(j);
        p=p->next;
    }
}
```

⑶ 广度优先遍历算法如下：

广度优先遍历算法 BFSTraverse

```
template <class T>
void ALGraph::BFSTraverse(int v)
{
   front=rear=-1;     //初始化队列, 假设队列采用顺序存储且不会发生溢出
   cout<<adjlist[v].vertex;   visited[v]=1; Q[++rear]=v;     //被访问顶点入队
   while（front!=rear）
   {
       v=Q[++front];
       p=adjlist[v].firstedge;       //边表中的工作指针 p 初始化
       while（p）
       {
          j= p->adjvex;
          if（visited[j]==0）{
              cout<<adjlist[j].vertex; visited[j]=1;Q[++rear]=j;
          }
          p=p->next;
       }
    }
}
```

4. 实验程序

参见光盘。

6.2 设计实验

6.2.1 求无向连通图的生成树

1. 问题描述

求无向连通图的一棵生成树。

2. 基本要求

⑴ 采用邻接矩阵存储；

⑵ 求深度优先生成树；

⑶ 输出该生成树的每一条边。

3. 设计思想

在一个连通无向图 $G=(V, E)$中，如果从任一个顶点开始进行深度优先遍历，必定将边集 E 分成两个集合 T 和 B，其中 T 是遍历过程中经历的边的集合，B 是剩余的边的集合。显然，边集 T 和图 G 中所有顶点一起构成连通图 G 的的一棵生成树。因此，修改 6.1.1 中邻接矩阵类 MGraph 的深度优先遍历算法，输出遍历所经过的边，算法如下：

深度优先生成树算法 DFSTraverse

```
template <class T>
void MGraph::BFSTraverse(int v)
{
    visited [v]=1;
    for (j=0; j<vertexNum; j++)
        if (arc[v][j]==1 && visited[j]==0) {
                                          cout<<"("<<v<<j<<")";
                                          DFSTraverse( j );
        }
}
```

【思考题】如何求广度优先生成树？

6.3 综合实验

6.3.1 TSP 问题

1. 问题描述

所谓 TSP 问题是指旅行家要旅行 n 个城市，要求各个城市经历且仅经历一次，并要求所走的路程最短。该问题又称为货郎担问题、邮递员问题、售货员问题，是图问题中最广为人知的问题。

2. 基本要求

⑴ 上网查找 TSP 问题的应用实例；

⑵ 分析求 TSP 问题的全局最优解的时间复杂度；

⑶ 设计一个求近似解的算法；

⑷ 分析算法的时间复杂度。

3. 设计思想

对于 TSP 问题，一种最容易想到的也肯定能得到最佳解的算法是穷举法，即考虑所有可能的旅行路线，从中选择最佳的一条。但是用穷举法求解 TSP 问题的时间复杂度为$O(n!)$，当 n 大到一定程度后是不可解的。

本实验只要求近似解，可以采用贪心法求解：任意选择某个城市作为出发点，然后前往最近的未访问的城市，直到所有的城市都被访问并且仅被访问一次，最后返回到出发点。

为便于查找离某顶点最近的邻接点，可以采用邻接矩阵存储该图。算法用伪代码描述如下：

伪代码

```
1. 任意选择某个顶点 v 作为出发点；
2. 执行下述过程，直到所有顶点都被访问：
    2.1 v=最后一个被访问的顶点；
    2.2 在顶点 v 的邻接点中查找距离顶点 v 最近的未被访问的邻接点 j；
    2.2 访问顶点 j；
3. 从最后一个访问的顶点直接回到出发点 v；
```

【思考题】上网查找 TSP 问题的应用实例，写一篇综述报告。

第 7 章 查找技术

查找又称检索，是数据处理中常用的一种重要操作。本章的实验内容针对各种查找技术展开，深刻理解并掌握基于不同查找结构的查找技术，在实际应用中遇到查找问题时，才能够灵活选择（或设计）合适的查找方法。

7.1 验证实验

7.1.1 顺序查找验证

1. 实验目的

⑴ 掌握顺序查找算法的基本思想；

⑵ 掌握顺序查找算法的实现方法；

⑶ 掌握顺序查找算法的时间性能。

2. 实验内容

对给定的数组（假设长度为 n），查找数组中与给定值 k 相等的元素。

3. 实现提示

顺序查找在 2.1.1 中已经做过实验，这里实现一种改进的顺序查找算法，即将待查值放在查找方向的“尽头”处，起哨兵作用，这样，就免去了在查找过程中每一次比较后都要判断查找位置是否越界，从而提高查找速度。

查找技术是通过平均比较长度来衡量查找性能的，为了统计在顺序查找过程中元素的比较次数，设置一个计数器 count 来记载比较次数，算法如下：

顺序查找算法

```
int SeqSearch1(int r[ ], int n, int k)
{
    r[0]=k ; count=0;
    i=n;
    while（++count && r[i]!=k）
        i--;
    cout<<"比较次数是："<<count;
    return i;
  }
```

4. 实验程序

参见光盘。

7.1.2 折半查找验证

1. 实验目的

⑴ 掌握折半查找算法的基本思想；

⑵ 掌握折半查找算法的实现方法；

⑶ 掌握折半查找算法的时间性能。

2. 实验内容

对给定的有序数组（假设长度为 n），查找数组中与给定值 k 相等的元素。

3. 实现提示

折半查找的基本思想为：在有序数组中，取中间元素作为比较对象，若给定值与中间元素相等，则查找成功；若给定值小于中间元素，则在中间元素的左半区继续查找；若给定值大于中间元素，则在中间元素的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无元素，查找失败。为了统计在折半查找过程中元素的比较次数，设置一个计数器 count 来记载比较次数，算法如下：

折半查找非递归算法 BinSearch1

```
int BinSearch1(int r[ ], int n, int k)
{
   low=1; high=n; count=0;
   while (low<=high)
   {
      mid=(low+high)/2;
      count++;
      if (k<r[mid]) high=mid-1;
            else if (k>r[mid]) low=mid+1;
                        else {
                              cout<<"比较次数是："<<count;
                              return mid;
                        }
   }
   cout<<"比较次数是："<<count;
   return 0;
}
```

4. 实验程序

参见光盘。

7.1.3 二叉排序树的建立

1. 实验目的

⑴ 掌握二叉排序树定义和特性；

⑵ 掌握二叉排序树的建立方法；

⑶ 实现基于二叉排序树的查找技术；

⑷ 掌握二叉排序树的查找性能。

2. 实验内容

⑴ 对给定的一组无序序列，建立一棵二叉排序树；

⑵ 对建立的二叉排序树实现查找操作。

3. 实现提示

二叉排序树通常采用二叉链表的形式进行存储，其结点结构定义如下（本章假定数据域均为整数）：

```
struct BiNode
{
   int data;
   BiNode *lchild, *rchild;
};
```

设计实验用二叉排序树类 BiSortTree，包括插入和查找操作。

```
class BiSortTree
{
 public:
    BiSortTree(int a[ ], int n);          //建立查找集合 a[n]的二叉排序树
    ~ BiSortTree( );          //析构函数，释放二叉排序树中所有结点，同二叉链表的析构函数
    void InsertBST(BiNode *root, BiNode *s); //在二叉排序树中插入一个结点 s
    BiNode *SearchBST(BiNode<int> *root, int k);    //查找值为 k 的结点
 private:
    BiNode *root;     //二叉排序树（即二叉链表）的根指针
};
```

⑴ 设计构造函数，即构造一棵二叉排序树的二叉链表存储，其过程是从空的二叉排序树开始，依次插入一个个结点。

在二叉排序树中插入结点的算法如下：

二叉排序树插入算法

```
void BiSortTree::InsertBST(BiNode *root, BiNode *s)
{
    if (root==NULL) root=s;
        else if (s->data<root->data) InsertBST(root->lchild, s);
                else InsertBST(root->rchild, s);
}
```

二叉排序树的构造函数算法如下：

二叉排序树构造函数算法

```
BiSortTree::BiSortTree(int r[ ], int n)
{
   for (i=0; i<n; i++)
   {
       s=new BiNode; s->data=r[i];
       s->lchild=s->rchild=NULL;
       InsertBST(root, s);
   }
}
```

⑵ 设计查找函数，在二叉排序树上进行查找的过程是一个递归的过程，算法如下：

二叉排序树查找算法 SearchBST

```
BiNode * BiSortTree::SearchBST(BiNode *root, int k)
{
    if (root==NULL) return NULL;
      else if (root->data==k) return root;
              else if (k<root->data) return SearchBST(root->lchild, k);
                      else return SearchBST(root->rchild, k);
}
```

4. 实验程序

参见光盘。

7.2 设计实验

7.2.1 顺序查找与折半查找的性能比较

1. 问题描述

对于给定的线性表要求用顺序查找和折半查找两种方法进行查找，并进行性能比较。

2. 基本要求

⑴ 对于有序数组，修改 7.1.1 中的顺序查找算法，利用数组的有序性提高查找性能；

⑵ 将 7.1.1 中的顺序查找算法和你设计的改进算法进行比较；

⑶ 对于同样的数组元素，同样的待查值，将顺序查找算法和折半查找算法进行比较；

⑷ 总结实验结果，给出不同算法进行比较的具体结论

3. 设计思想

对于给定的数组采用顺序查找时，查找失败的平均查找长度为 $O(n)$。但对有序数组进行顺序查找，可减少查找失败的平均查找长度，因为不需要遍历整个表就能确定表中不存在要查找的记录。具体算法如下：

改进的顺序查找算法

```
int SeqSearch2(int r[ ], int n, int k)
{
    r[0]=k; count=0;
    i=n;
    while (++count && r[i]>=k)
          i--;
    cout<<"比较次数是："<<count;
    if (r[i]= =k && i!=0) return i;
    else return 0;
}
```

验所设计的顺序查找和折半查找的算法以及上述改进的顺序查找算法，比较三种方法的时间性能（即待查值与数组元素的比较次数）。

【思考题】可以有多种方法提高顺序查找的时间性能，例如将数组元素按访问频度递减的顺序存放，

请设计并实现一种改进的顺序查找算法。

7.3 综合实验

7.3.1 简单个人电话号码查询系统

1. 问题描述

人们在日常生活中经常需要查找某个人或某个单位的电话号码，本实验将实现一个简单的个人电话号码查询系统，根据用户输入的信息（例如姓名等）进行快速查询。

2. 基本要求

⑴ 在外存上，用文件保存电话号码信息；

⑵ 在内存中，设计数据结构存储电话号码信息；

⑶ 提供查询功能：根据姓名实现快速查询；

⑷ 提供其他维护功能：例如插入、删除、修改等。

3. 设计思想

由于需要管理的电话号码信息较多，而且要在程序运行结束后仍然保存电话号码信息，所以电话号码信息采用文件的形式存放到外存中。在系统运行时，需要将电话号码信息从文件调入内存来进行查找等操作，为了接收文件中的内容，要有一个数据结构与之对应，可以设计如下结构类型的数组来接收数据：

```
const int max=10;
struct TeleNumber
{
    string name;     //姓名
    string phoneNumber;      //固定电话号码
    string mobileNumber;     //移动电话号码
    string email;     //电子邮箱
} Tele[max];
```

为了实现对电话号码的快速查询，可以将上述结构数组排序，以便应用折半查找，但是，在数组中实现插入和删除操作的代价较高。如果记录需频繁进行插入或删除操作，可以考虑采用二叉排序树组织电话号码信息，则查找和维护都能获得较高的时间性能。更复杂地，需要考虑该二叉排序树是否平衡，如何使之达到平衡。有关折半查找和二叉排序树的具体算法请参见主教材相关内容。

第 8 章 排序

排序是计算机程序设计中的一种重要操作，其主要目的是为了提高查找效率。本章的实验内容是各种排序方法的实践，深刻理解并掌握各种排序方法，在实际应用中，才能选择（或设计）最合适的排序方法或几个排序方法的组合解决排序问题。

8.1 验证实验

8.1.1 直接插入排序算法验证

1. 实验目的

⑴ 掌握直接插入排序算法的基本思想；

⑵ 掌握直接插入排序算法的实现方法；

⑶ 验证直接插入排序算法的时间性能。

2. 实验内容

对一组数据进行直接插入排序（按升序排列）。

3. 实现提示

直接插入排序的基本思想是：依次将待排序序列中的每一个记录插入到一个已排好序的序列中，直到全部记录都排好序。具体的排序过程是：

⑴ 将整个待排序的记录序列划分成有序区和无序区，初始时有序区为待排序记录序列中的第一个记录，无序区包括所有剩余待排序的记录；

⑵ 将无序区的第一个记录插入到有序区的合适位置中，从而使无序区减少一个记录，有序区增加一个记录；

⑶ 重复执行⑵，直到无序区中没有记录为止。

不失一般性，假定待排序的数据存储在一维数组 r[n]中，且数组元素为整数。为验证直接插入排序算法的时间性能，在排序过程中，设置两个计数器 count1 和 count2，分别用来统计元素的比较次数和移动次数。算法如下：

直接插入排序算法 InsertSort

```
void InsertSort(int r[ ], int n)
{
    count1=0; count2=0;
    for (i=2; i<=n; i++)
    {
        r[0]=r[i];        //设置哨兵
        for (j=i-1; ++count1&& r[0]<r[j]; j--)      //寻找插入位置
        {
            r[j+1]=r[j];                    //记录后移
            count2++;
        }
        r[j+1]=r[0];
    }
    cout<<"比较次数为："<<count1<<endl;
    cout<<"移动次数为："<<count2+2<<endl;     //加上两个赋值语句
}
```

4. 实验程序

参见光盘。

8.1.2 起泡排序算法验证

1. 实验目的

⑴ 掌握起泡排序算法的基本思想；

⑵ 掌握起泡排序算法的实现方法；

⑶ 验证起泡排序算法的时间性能。

2. 实验内容

对一组数据进行起泡排序（按升序排列）。

3. 实现提示

起泡排序的基本思想是：两两比较相邻记录的关键码，如果反序则交换，直到没有反序的记录为止。具体的排序过程为：

⑴ 将整个待排序的记录序列划分成有序区和无序区，初始状态有序区为空，无序区包括所有待排序的记录。

⑵ 对无序区从前向后依次将相邻记录的关键码进行比较，若反序则交换，从而使得关键码小的记录向前移，关键码大的记录向后移（像水中的汽泡，体积大的先浮上来）。

⑶ 重复执行⑵，直到无序区中没有反序的记录。

不失一般性，假定待排序的数据存储在一维数组 r[n]中，且数组中的每个元素都是整数。为验证起泡排序算法的时间性能，在排序过程中，设置两个计数器 count1 和 count2，分别用来统计元素的比较次数和移动次数。算法如下：

起泡排序算法 BubbleSort

```
void BubbleSort(int r[ ], int n)
{
   count1=0; count2=0;
   exchange=n;                  //第一趟起泡排序的范围是 r[1]到 r[n]
   while (exchange)             //仅当上一趟排序有记录交换才进行本趟排序
   {
      bound=exchange; exchange=0;
      for (j=1; j<bound; j++)                  //一趟起泡排序
         if (++count1 && r[j]>r[j+1]) {
                        r[j]←→r[j+1];
                        count2+=3;     //一次交换需要三个赋值语句
                        exchange=j;    //记录每一次发生记录交换的位置
         }
   }
   cout<<"比较次数为："<<count1<<endl;
   cout<<"移动次数为："<<count2<<endl;
}
```

4. 实验程序

参见光盘。

8.1.3 简单选择排序算法验证

1．实验目的

⑴ 掌握简单选择排序算法的基本思想；

⑵ 掌握简单选择排序算法的实现方法；

⑶ 验证简单选择排序算法的时间性能。

2. 实验内容

对一组数据进行简单选择排序（按升序排列）。

3. 实现提示

简单选择排序的基本思想是：第 i 趟排序通过 $n-i$ 次关键码的比较，在 $n-i+1$（$1\le i\le n-1$）个记录中选取关键码最小的记录，并和第 i 个记录交换作为有序序列的第 i 个记录。具体的排序过程为：

⑴ 将整个记录序列划分为有序区和无序区，初始状态有序区为空，无序区含有待排序的所有记录。

⑵ 在无序区中选取关键码最小的记录，将它与无序区中的第一个记录交换，使得有序区扩展了一个记录，而无序区减少了一个记录。

⑶ 不断重复⑵，直到无序区只剩下一个记录为止。此时所有的记录已经按关键码从小到大的顺序排列就位。

不失一般性，假定待排序的数据存储在一维数组 r[n]中，且数组中的每个元素都是整数。为验证简单选择排序算法的时间性能，在排序过程中，设置两个计数器 count1 和 count2，分别用来统计元素的比较次数和移动次数。算法如下：

简单选择排序算法 SelectSort

```
void SelectSort(int r[ ], int n)
{
   count1=0; count2=0;
   for (i=1; i<n; i++)          //对 n 个记录进行 n-1 趟简单选择排序
   {
       index=i;
       for (j=i+1; j<=n; j++)     //在无序区中选取最小记录
          if (++count1 && r[j]<r[index]) index=j;
       if (index!=i) {
                       r[i]←→r[index];
                       count2+=3;     //一次交换需要三个赋值语句
       }
   }
   cout<<"比较次数为："<<count1<<endl;
   cout<<"移动次数为："<<count2<<endl;
}
```

4. 实验程序

参见光盘。

8.2 设计实验

8.2.1 直接插入排序基于单链表的实现

1. 问题描述

采用单链表存储待排序数据，实现直接插入排序算法。

2. 基本要求

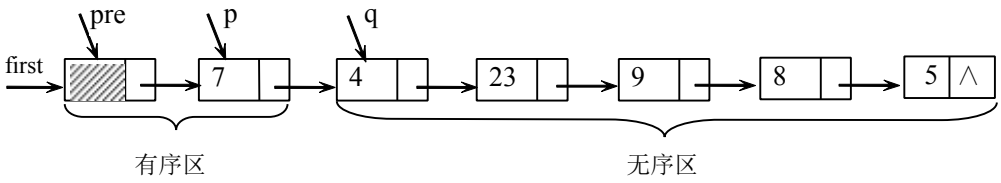
⑴ 采用单链表存储待排序数据；

⑵ 设计直接插入排序算法；

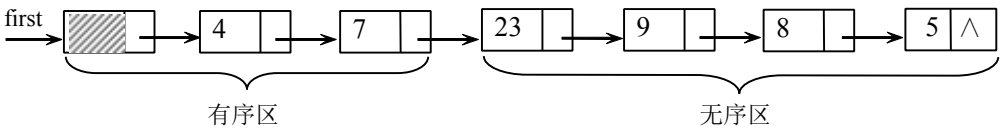
⑶ 与数组存储下直接插入排序算法进行对比。

3. 设计思想

首先将待排序数据建立一个带头结点的单链表，具体算法请参见 2.1.2 自行完成。在单链表中进行直接插入排序的基本思想是：将单链表划分为有序区和无序区，有序区只包含一个元素结点，依次取无序区中的每一个结点，在有序区中查找待插入结点的插入位置，然后把该结点从单链表中删除，再插入到相应位置。例如，有一组待排序数据存储在单链表 first 中，排序过程如下：



(a) 待排序记录划分为有序区和无序区



(b) 将 4 插在 7 的前面，有序区多了一个结点

图 8-2　直接插入排序过程示例

分析上述排序过程，需要设一个工作指针 q 在无序区中指向待插入的结点，为了查找正确的插入位置，每趟排序前需将工作指针 pre 和 p 指向头结点和开始结点，在找到插入位置后，将结点 q 插在结点 pre 和 p 之间。这相当于在单链表中删除结点 q，因此为了保证链表不断开，需要在删除结点 q 之前保留结点 q 的后继结点的地址。算法如下：

直接插入排序算法

```
void  StraightSort(Node *first)
{
  pre=first; p=first->next; q=p->next;
  while (q)
  {
     while (q!=p)
     {
        while (p->data<q->data)
        {
           pre=p;
           p=p->next;
        }
        if (p!=q) {
                  u=q->next
                  pre->next=q;
                  q->next=p;
                  q=u;
        }
        else q=q->next;
     }
     pre=first; p=first->next;
  }
}
```

【思考题】扩充该实验的思想，总结基于单链表存储下各种排序算法的实现。

8.3 综合实验

8.3.1 各种排序算法时间性能的比较

1. 问题描述

对本章的各种排序方法（直接插入排序、希尔排序、起泡排序、快速排序、直接选择排序、堆排序和归并排序）的时间性能进行比较。

2. 基本要求

⑴ 设计并实现上述各种排序算法；

⑵ 产生正序和逆序的初始排列分别调用上述排序算法，并比较时间性能；

⑶ 产生随机的初始排列分别调用上述排序算法，并比较时间性能。

3. 设计思想

上述各种排序方法都是基于比较的内排序，其时间主要消耗在排序过程中进行的记录的比较次数和移动次数，因此，统计在相同数据状态下不同排序算法的比较次数和移动次数，即可实现比较各种排序算法的目的。

直接插入排序、起泡排序、直接选择排序在 8.1 中已经实现，请仿照 8.1 中的方法在其他排序算法中的适当位置插入计数器统计元素的比较次数和移动次数。

【思考题】如果测算每种排序算法所用实际的时间，应如何修改排序算法？