附录 1. 关于 C++程序设计

在程序开发过程中通常需要做到如下两点：一是高效地描述数据；二是设计一个好的算法，该算法最终可用程序来实现。要想高效地描述数据，必须具备数据结构领域的专门知识;要想设计一个好的算法,需要算法设计领域的专门知识。这里我们采用 C++语言来描述算法，并假定大家已经掌握了 C++程序设计的基本方法。同时还假设大家已经具备数据结构的必备知识，剩下的主要任务是探讨算法设计领域的专门知识。为了便于今后的学习，我们还是简要地回顾一下 C++程序设计方面的有关知识和数据结构方面的知识。本章主要复习 C++程序设计方面的知识，内容主要有：

模板函数、参数传递方式、类与模板类、类的共享成员－保护成员－私有成员、友元、操作符重载、数组的动态分配与释放等。

§1 模板函数

1.1 函数与模板函数

考察下列函数：

程序 1-1 计算一个整数表达式

```
int Abc(int a, int b, int c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
}
```

程序 1-2 计算一个浮点数表达式

```
float Abc(float a, float b, float c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
```

```
}
```

这两个函数中除了形式参数与返回值的类型不同外，再没有别的差别。如果我们能够编写一段通用代码，将参数的数据类型作为一个变量，它的值由编译器来确定，就可以简化程序的编写工作，这一特性在C++语言的模板函数中体现。

程序 1-3 利用模板函数计算一个表达式

```
template <class T>
T Abc(T a, T b, T c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
}
```

利用这段通用代码，通过把 T 替换成 int，编译器可以立即构作出程序 1-1；把 T 替换成 float 又可以立即构作初程序 1-2。同样可以把 T 替换成 double 和 long，等等。把函数 Abc 编写成模板函数可以使我们免去了解形式参数的类型。

1.2 传值参数和引用参数

在上面的函数 Abc 中，a，b，c 是形式参数。如果调用函数 Abc 时采用

```
z=Abc(x,y,2)
```

x，y，2 分别是对应 a，b，c 的实际参数。a，b，c 都是传值参数。运行时，与传值形式参数对应的实际参数的值将在函数执行之前被复制给形式参数，复制过程是由该形式参数所属数据类型的复制构造函数（copy constructor）完成的。如果实际参数与形式参数的数据类型不同，必须进行类型转换，从实际参数的类型转换为形式参数的类型（这里假定该种类型转换是允许的）。当函数运行结束时，形式参数所属数据类型的析构函数（destructor）负责释放该形式参数。当一个函数返回时，形式参数不会被复制到对应的实际参数中。因此，函数调用不会修改实际参数的值。

值得注意的是，采用传值参数会增加程序的开销。假定 a，b，c 是传值参数，在函数 Abc 被调用时，类型 T 的复制构造函数把相应的实际参数分别复制到形式参数 a,b,c 之中，以供函数 Abc 使用。如果用具有 100 个元素的矩阵 Matrix 作为实际参数来调用函数 Abc，那么复制三个实际参数给 a，b，c，将需要 300 次操作。同理，当函数 Abc 返回时，其析构函数又需要花费额外的 300 次操作来释放 a，b，c。

C++语言通过使用引用参数来减少这笔开销。

程序 1-4 利用引用参数计算一个表达式

```
template<class T>
T Abc(T& a, T& b, T& c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
}
```

在程序 1-4 中，a，b，c，是引用参数（reference parameter）。如果用语句 Abc（x，y，z）来调用函数 Abc，其中 x，y，z 是相同的数据类型，那么这些实际参数被分别赋予名称 a，b，c，因此，在函数 Abc 执行期间，x，y，z 被用来替换对应的 a，b，c。与传值情况不同，在函数调用时，本程序并没有复制实际参数的值，在返回也没有调用析构函数。如果 x，y，z 分别是 100 个元素的矩阵，由于不需要把 x，y，z 的值复制给对应的形式参数，因此，我们至少可以节省采用传值参数进行复制时所需要的 300 次操作。

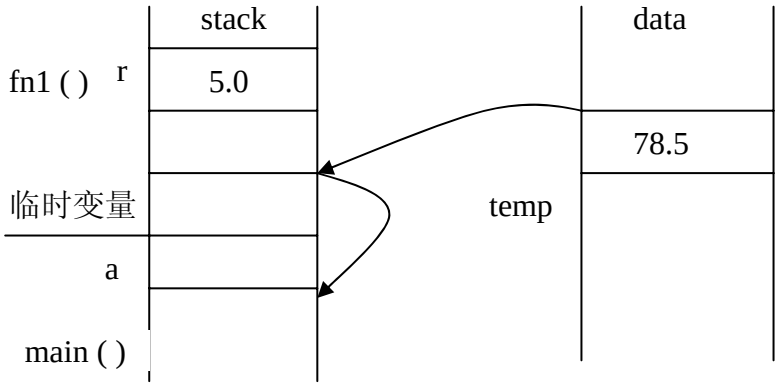
事实上，引用只是个别名，当建立引用时，程序用另一个变量或对象（目标）的名字初始化它。引用不是值，不占用存储空间，声明引用时，目标的存储状态不会改变。引用一旦初始化，它就维系在一定的目标上，再也不分手。

函数的返回值也可以采取引用的方式。

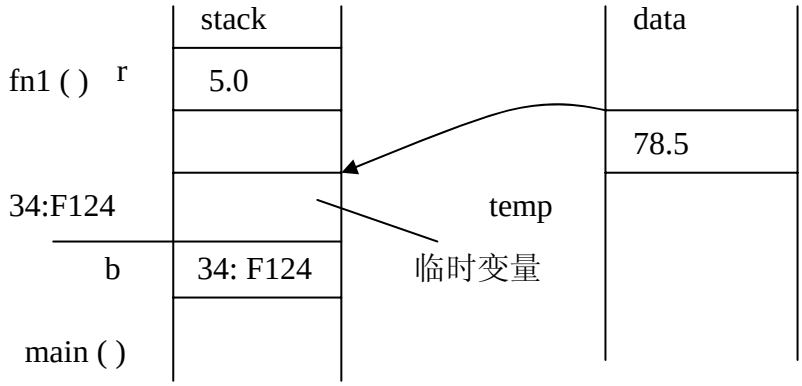
程序 1-6 关于函数的返回值或返回引用的程序

```
# include <iostream.h>
float temp;
float fn1(float r)
{
  temp=r*r*3.14;
  return temp;
}
float& fn2(float r)
{
  temp=r*r*3.14;
  return temp;
}
void main()
  float a=fn1(5.0);  //形式 1
  float& b=fn1(5.0);  //形式 2
  float c=fn2(5.0);  //形式 3
  float& d=fn2(5.0);  //形式 4
   cout <<a <<endl;
   cout <<b <<endl;
   cout <<c <<endl;
   cout <<d <<endl;
}
```

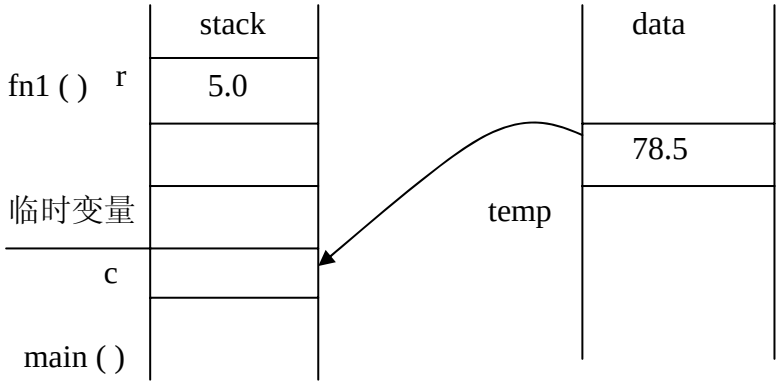
对于主函数的四种返回方式，运行结果虽是一样的，但它们在内存中的活动情况各不相同。其中，变量 temp 是全局数据，驻留在全局数据区 data，函数 main（）、函数 fn1（）及函数 fn2（）驻留在栈区 stack。以下四种返回形式的表示：



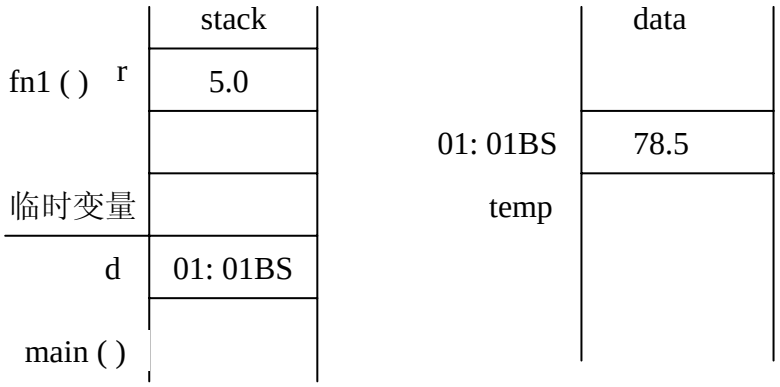
1． 传值参数和返回值方式内存布局



2． 返回值初始引用的内存布局



3． 返回引用方式的内存布局



4. 返回引用方式的值作为引用的初始化

采用引用参数或返回引用应充分考虑到被引用变量的生命期，免得出现数据失效现象。

§1.2 动态存储分配与异常引发

C++操作符 new 可用来进行动态存储分配，该操作符返回一个指向所分配空间的指针。例如在使用数组时，很多情况下数组的大小在编译时可能是未知的。事实上，它们可能是随着函数调用的变化而变化。对于一个大小为 n 的一维浮点数组可以按如下方式来创建：

```
float *x=new float [n];
```

构建二维数组时，可以把它看成是由若干行组合起来的，每一行都是一个一维数组，动态存储时需多次调用 new。

程序 1-7 为二维数组动态分配空间

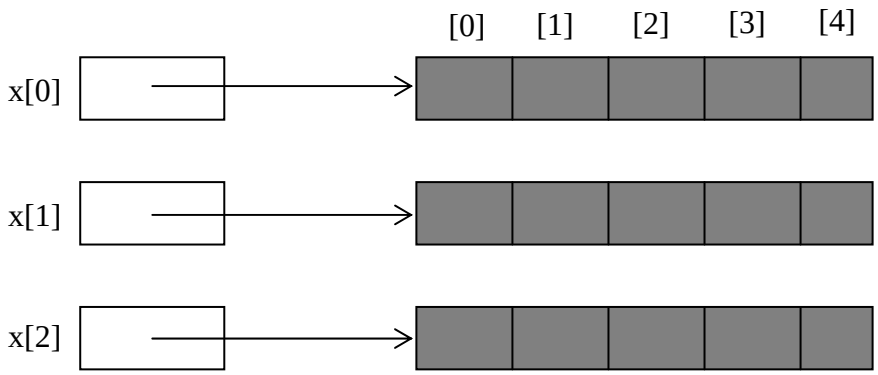
```
template<class T>
bool Make2Darray(T** &x, int rows, int cols)
{//创建一个二维数组
```

```
        try{//创建指针
          x=new T* [rows];
          //为每一行分配空间
          for (int i=0; i<rows; i++)
          x[i]=new int [cols];
         return true;
       }
      catch(xalloc){return false;}
}
```

这里 rows 是数组的行数，cols 是数组的列数。语句 catch（xalloc）{return false;}是引发异常，当空间分配失败时，返回 false。

程序 1-8 释放由 Make2Darray 所分配的空间

```
template<class T>
void Delete2Darray(T** &x, int rows)
{//删除二维数组 x
 //释放为每一行所分配的空间
  for(int i=0; i<rows; i++)
    delete [] x[i];
  //删除行指针
  delete [] x;
  x=0;
}
```



一个 3×5 数组的存储结构

§1.3 C++的类与类模板

类构成了实现 C++面向对象程序设计的基础。类是 C++封装的基本单元，它把数据和函数封装在一起。当类的成员声明为保护时，外部不能访问；声明为公共时，则在任何地方都可以访问。将类定义和其成员函数定义分开，是目前开发程序的通常做法。我们可以把类定义（头文件）看成是类的外部接口，类的成员函数定义看成是类的内部实现。定义了一个类以后，可以把该类名作为一种数据类型，定义其“变量”（对象）。

类中有两个特殊的成员函数：构造函数与析构函数。构造函数创建类对象，初始化其成员，析构函数撤销类对象。这两个函数成员的设计和应用，直接影响编译程序处理对象的方式，它们的实现使 C++的类机制得以充分的体现。

例子 类 Currency 的建立

类 Currency 的对象主要拥有三个成员：符号（sign）、美元（dollars）、美分（cents）。如$2.35, sign＝＋，dollars＝2，cents＝35；-$6.05, sign＝－，dollars＝6，cents＝5。对于这种类型的对象，主要执行的操作是：

1）、设置成员的值；

2）、确定各成员的值；

3）、增加两种货币的类型；

4）、增加成员的值；

5)、输出。

首先确定用无符号的长整型变量 dollars、无符号整型变量 cents 和 sign 类型的变量 sgn 来描述货币对象，其中 sign 类型的定义为：

```
enum sign{plus, minus};
```

程序 1-9 定义 Currency 类

```
class Currency{
  public
  // 构造函数
  Currency(sign s=plus, unsigned long d=0, unsigned int c=0);
  // 析构函数
  ～Currency() {}
  bool Set(sign s, unsigned long d=0, unsigned int c=0);
  bool Set(float a);
  sign Sign() const {return sgn;}
  unsigned long Dollars() const {return dollars;}
  unsigned int Cents() const {return cents;}
  Currency Add(const Currency& x) const;
  Currency& Increment(const Currency& x);
  Void Output() const;
private:
  sign sgn;
  unsigned long dollars;
  unsigned int cents;
}
```

说明：public 部分的第一个函数与 Currency 类同名，即是构造函数，其指明如何创建一个类型的对象，它不可有返回值。此处，构造函数有三个参数。在创建一个类对象时，构造函数被自动唤醒。如

```
Currency f, g(plus,3.45), h(minus,10);
Currency *m=new Currency(plus,8,12);
```

分别创建了“变量”对象和一个指向 Currency 对象的指针（变量）对象。但是，在程序 1-9 中，类的成员函数只有声明，未有描述(或实现)，因而还必须在类定义的外部给出各个成员函数的描述。

程序 1-9.1 Currency 的构造函数

```
Currency::Currency(sign s, unsigned long d, unsigned int c)
{// 创建一个 Currency 对象
  if(c>99)
    {//美分数目过多
     cerr<<”Cents should be < 100” << endl;
     exit(1);}
  sgn=s; dollars=d; cents=c;
}
```

构造函数在初始化当前对象的 sgn、dollars、cents 等数据成员之前需要验证参数的合法性。

程序 1-9.2 设置 private 数据成员

```
bool Currency::Set(sign s, unsigned long d, unsigned int c)
{// 取值
  if(c>99) return false;
  sgn=s; dollars=d; cents=c;
```

```
  return true;
}
bool Currency::Set(float a)
{// 取值
  if(a<0) {sgn=minus; a=-a;}
  else sgn=plus;
  dollars=a; //抽取整数部分
  // 获取两个小数位
  cents=(a+0.005-dollars)*100;
  return true;
}
```

程序 1-9.3 累加两个 Currency

```
Currency Currency::Add(const Currency& x) const
{// 把 x 累加到 *this
  long a1, a2, a3;
  Currency ans;
  // 把当前对象转换成带符号的整数
  a1=dollars * 100 + cents;
  if(sgn==minus) a1=-a1;
  // 把 x 转换成带符号的整数
  a2=x.dollars * 100 + x.cents;
  if(x.sgn==minus) a2=-a2;
  a3=a1 + a2;
```

```
    // 转换成 currency 形式
    if(a3<0) {ans.sgn=mins; a3=-a3;}
    else ans.sgn=plus;
    ans.dollars=a3/100;
    ans.cents=a3-ans.dollars * 100;
    return ans;
}
```

说明：在 C++中，保留关键字 this 用于指向当前对象，*this 代表对象本身。

程序 1-9.4　Increment 与 Output

```
Currency& Currency::Increment(const Currency& x)
{// 增加量 x
   *this=Add(x);
   return *this;
}
void Currency::Output() const
{// 输出 currency 的值
  if(sgn==minus) cout <<' - ';
  cout <<' $' <<dollars <<'.' ;
  if(cents <10) cout << "0" ;
  cout << cents;
}
```

说明：利用成员函数来设置数据成员的值可以确保成员拥有合法的值，如，

构造函数和 Set 函数已经做到这一点。其他成员当然也应该保证数据成员的合法性。因此，在诸如 Add 和 Output 函数的代码中就不必再验证了。

程序 1-10 Currency 类应用示例

```
#include <iostream.h>
#include "curr1.h"
void main(void)
{
   Currency g, h(plus, 3, 50), i, j;
   g.Set(minus, 2, 25);
  i.Set(-6.45);
   j=h.Ad(g);
   j.Output(); cout << endl;
   i.Increment(h);
   i.Output(); cout << endl;
   j=i.Add(g).Add(h);
   j.Output(); cout << endl;
   i.Output(); cout << endl;
}
```

说明：程序 1-10 的第一行定义了四个Currency类变量：g，h，i和j。除h具有初值$3.50 外，构造函数把它们都初始化位$0.00.接下来的两行将g和I分别设置成-$2.25 和-$6.45,之后调用函数Add把g和h加在一起，并把返回的对象（值为$1.25）付给j。为此，需要使用缺省的赋值过程把右侧对象的各数据分别复制到相应的数据成员，复制的结果是使j具有值$1.25,此值在下一行被输出。再下一句使i增加一个量h，然后输出i的新值（-$2.95）。语句j=i.Add(g).Add(h);表示首先将i与g相加，返回一个临时变量（值为-$5.20）,然后把这个临时变量与h相加，并返回另一个临时变量（值为-$1.70）,最后，将新的临时变量复制到

j中。这里，’.’序列的处理顺序是从左至右的。

§1.4 重载与友元

严格来讲，一个函数的返回值、参数及“函数规则”三个因素就确定了该函数。有一个因素不同，就认为是不同的函数，尽管它们有相同的名字。函数的重载正是利用了“参数的变化或还回值的变化”而得到的（属于不同类的函数认为是不同的）。为了使程序简明，C++程序中采用“重载”是常见的。

在 Currency 类中包含了几个与 C++标准操作相类似的成员函数，如，Add 进行 ＋ 操作，Increment 进行 ＋＝ 操作等。直接使用这些标准的操作符比另外定义新的函数（如 Add 和 Increment）要自然得多。可以借助操作符重载的过程来使用＋和＋＝。操作符重载允许扩充 C++操作符的功能，以便包它们直接应用到新的数据类型或类中。

程序 1-11.1 使用操作符重载的 Currency 类定义

```
class Currency {
  public:
  //构造函数
  Currency(sign s=plus, unsigned long d=0, unsigned int c=0);
  //析构函数
  ~Currency() {}
  bool Set(sign s, unsigned long d, unsigned int c);
  bool Set(float a);
  sign Sign() const
    {if (amount < 0) return minus;
     else return plus;}
  unsigned long Dollars() const
   {if (amount < 0) return (-amount)/100;
    else return amount/100;}
```

```
    unsigned int Cents() const
      {if (amount < 0)
        return  -amount - Dollars()*100;
       else return amount  -  Dollars()*100;}
    Currency operator+(const Currency& x) const;
    Currency& operator+=(const Currency& x)
      {amount+=x.amount; return *this;}
    void Output(ostream& out) const;
  private:
    long amount;
};
```

程序 1-11.2 十，output 和 << 代码

```
 Currency Currency:: operator+(const Currency& x) const
{//把 x 累加到*this
  Currency y;
  y.amount=amount+x.amount;
}
void Currency::Output(ostream& out) const
{//将 currency 的值插入到输出流
  long a=amount;
  if (a<0) {out << '-'; a=-a;}
  long d=a/100; //美元
  out << '$' << d << '.';
```

```
  int c=a-d*100; //美分
  if (c<10) out << “0”;
  out << c;
}
//重载 <<
ostream& operator << (ostream& out, const Currency& x)
  {x.Output (out); return out;}
```

注：由于在前面的类定义中没有关于<< 的函数声明，所以，在程序 1-11.2 的关于操作符 << 的重载时，借用了类成员函数 Output。这是因为，函数

```
ostream& operator <<(ostream& out, const Currency& x)
```

不能访问 private 成员 amount。

事实上，一个类的 private 成员仅对于类的成员函数时可见，而在有些应用中，必须把对这些 private 成员的访问权授予其他的类和函数，一般的做法是把这些类和函数定义为友元。上面关于 << 的重载可以如下完成：

1. 在类 Currency 的定义中添加语句

```
friend ostream& operator << (ostream&, const Currency&);
```

2. 给出重载友元 << 程序

程序 1-11.3 重载友元 <<

```
// 重载 <<
ostream& operator << (ostream& out, const Currency& x)
{//把 currency 的值插入到输出流
   long a=x.amount;
   if (a<0) {out << ‘- ‘; a=-a;}
   long d=a/100; //美元
   out << ‘$’ << d << ‘.’;
```

```
    int c=a-d*100; //美分
    if (c<10) out << “0”;
    out << c;
    return out;
}
```

利用操作符重载，程序 1-10 可以改写如下，看起来更自然、可读。

程序 1-11.3 操作符重载的应用

```
# include <iostream.h>
# include “curr3.h”
void main(void)
{
  currency g, h(plus, 3, 50), i, j;
  g.Set(minus, 2, 25);
   i.Set(-6.45);
   j=h+g;
   cout << j << endl;
   i+=h;
   cout << i <<endl;
   j=i+g+h;
   cout << j << endl;
   j=(i+=g)+h;
   cout << j << endl;
   cout << i <<endl;
}
```

§1.5 检测与调试

检查程序的正确性，主要是数学证明方法和程序测试过程。很多情况下，用数学证明的方法，在后面要提到。但是，多数情况下用数学证明方法证明一个程序的正确性是十分困难的，这里谈一下程序测试过程。所谓程序测试过程是指在目标计算机上利用输入数据，也称为测试数据，来实际运行该程序。把程序的实际行为与所期望的行为进行比较。如果两种行为不同，就可判定程序中有问题存在。值得注意的是只使用个别的数据进行测试而未发现问题，并不能说明程序本身是正确的。所以，用测试过程来说明程序的正确性必须有足够的测试数据。对于大多数实际的程序，可能的测试数据数量很大，不可能进行穷尽测试。为此，我们需要选取具有代表性的部分数据进行测试。实际用来测试的输入数据空间的子集称为测试集。

例 求解二次方程

$$ax^2 + bx + c = 0$$

其中a,b,c都是已知数，$a \neq 0$.我们用如下程序来求解：

程序 1-12　计算并输出一个二次方程的根

```
template < class T >
void OutputRoots( T a, T b, T c )
{//计算并输出一个二次方程的根
  T d = b*b-4*a*c;
  if ( d > 0 ) {// 两个实根
    float sqrtd = sqrt (d );
    cout << “There are two real roots”
    << (-b+sqrtd)/(2*a) << “and”
    << (-b+sqrtd)/(2*a)
    <<endl; }
```

```
    else if (d==0)
        // 两个根相同
        cout << “There is only one distinct root”
           << -b/(2*a)
           << endl;
    else // 复数根
       cout << “The roots are complex”
           << endl
           << “The real part is”
           << -b/(2*a) << endl
           << sqrt(-d)/(2*a) << endl;
}
```

如果采用测试过程来验证该程序的正确性，则所有可能的输入数据就是所有不同的三元组 (a,b,c)，其中 $a \neq 0$。即使 a,b,c 都被限制为整数，而且长度为 16 位，所有不同的三元组的数目将达到 $2^{32}(2^{16}-1)$。如果目标计算机能按每秒 100 万个三元组进行测试，也至少需要三天才能完成。所以，实际测试集仅是测试数据空间的子集。显然，使用测试子集所完成的测试不能保证程序的正确性。测试的目的不是去建立正确性认证，而是为了暴露程序中的错误。因此，必须选取能暴露程序中所存在的错误的测试数据。设计测试数据的技术分为两种：黑盒法和白盒法。

比较流行的黑盒法是 I/O 分类及因果图。在 I/O 分类方法中，输入数据和/或输出数据空间被分成若干类，不同类中的数据会使程序表现出的行为有本质的不同，而相同类中的数据则使程序表现出本质上类似的行为。如二次方程求解的例子，输入和/或输出数据有三种本质不同的行为：产生两个不同的实根、产生唯一的实根、产生复数根。据此，将测试数据空间分为三个类，一个测试集应至少从每个类中抽取一个输入数据。$\{(1,-5,6),(1,-8,16),(1,2,5)\}$ 是符合黑盒法的测试集。

白盒法基于对代码的考察来设计测试数据。对一个测试集最起码的要求是

使程序中的每条语句都至少执行一次，此称为语句覆盖。另有分支覆盖，其要求测试集能使程序中的每一个条件都能分别出现 true 和 false 两种情况。可以加强分支覆盖的要求，要求每个条件中的每个从句（不包含布尔操作符&&,||,!的布尔表达式）既能出现 true 又能出现 false 情况，此称为从句覆盖。如果取定一个测试数据，程序在执行时各个语句被执行可以按先后排出一个顺序，形成一个执行路径。不同的测试数据可能会得到不同的执行路径。在白盒法中，一般要求实现执行路径覆盖。但是，在实践中，一般不可能进行全部执行路径覆盖，因为所有可能的执行路径可能具有很大的数目。在这里，我们要求使用白盒法时，应至少达到语句覆盖。另外，在测试时还应特别注意容易使程序出错的特定情形（一般是边界情形）。

程序 1-13 寻找最大元素

```
template < class T >
int Max(T a[], int n)
{//寻找 a[0:n-1]中的最大元素
  int pos=0;
  for (i=1; i < n; i ++)
  if (a[pos] < a[i])
    pos = I;
  return pos;
}
```

对于程序 1-13，测试数据 a[0:4]=[2,4,6,8,9]能够提供语句覆盖，但不能提供分支覆盖，因为条件 a[pos]<a[i] 不会变成 false。测试数据[4,2,6,8,9]既能提供语句覆盖也能提供分支覆盖。在程序 1-12 中只存在三条执行路径：1～7 行；1、2、8～12 行；1、2、8、13～19 行。在程序 1-13 中，执行路径的条数随着 n 的增加而增加，以至能够达到任意多。

测试能够发现程序中的错误，确定并纠正程序错误的过程称为调试（debug）。几点建议：

1. 可用采用逻辑的方法来确定错误的语句。

2. 如果方法 1 失败，还可以采用程序跟踪的方法，以确定程序在何时出现错误。

3. 如果普遍跟踪不易做到（对于给定的测试数据，程序需要运行的语句或指令太多），则采用专门跟踪的办法，这要求首先将可以的代码分离出来。

4. 在测试和调试一个有错的程序时，从一个与其他函数独立的函数开始。这个函数最好是一个典型的输入或输出函数。然后每次引入一个尚未测试的函数，测试并调试更大一点的程序。此称为增量测试与调试。在使用这种策略时，有理由认为产生错误的语句位于刚刚引入的函数之中。

5. 不要试图通过产生异常来纠正错误。在纠正一个错误时，必须保证不会产生一个新的、以前没有的错误。用原本能使程序正确运行的测试数据来运行纠正过错误的程序，确信对于该数据，程序仍然正确。

附录 2

基本数据结构

改进算法的基本技术之一就是数据结构化，使得由此产生的操作得以高效的执行。本章介绍几种基本的数据结构，包括栈和队列、二叉树以及图的概念。

2.1 栈和队列

在计算机程序中最长用的数据组织形式是有序表和线性表，通常记做 $a=(a_1,a_2,\cdots,a_n)$，这里 a_i 称为元素，来源于某个集合。空表有 $n=0$ 个元素。栈就是一个有序表，其插入和删除操作都在称作栈顶的一端进行。队列同样也是一个有序表，其插入操作在队尾一端进行，而删除在头一端进行。栈也被称作后入先出表（LIFO），表示对其中元素的操作顺序，而队列也被称作先入先出（FIFO）表，表示先进入队列的元素先出队。表示栈最简单的方式是用一维数组，如 $\text{stack}[MaxSize]$，其中 $MaxSize$ 是该栈容纳元素的最大个数。栈中最底元素存储在 $\text{stack}[0]$，第 i 个元素存储在 $\text{stack}[i-1], i=1,2,\cdots$。用一个与数组相关的变量指向栈中最顶层的元素，该变量记做 top。要检验栈是否为空，考察 $\text{if }(\text{top}<0)$ 是否成立，如果不成立，则最顶层元素就在 $\text{stack}[\text{top}]$ 中；要检验栈是否已满，可以考察 $\text{if }(\text{top}>=(MaxSize-1))$ 是否成立。对栈的两个重要操作是插入（Add）和删除（Delete）元素。用 C++语言可以描述如下

程序 2-1-1 栈的类定义

```
include < iotream.h >
template < class Type >
class Stack
{
   private :
        int top, MaxSize ;
        Type    * stack ;
    public:
        Stack (int MSize ) : MaxSize ( MSize )
        { Stack = new Type [ MaxSize ] ; top = -1 ; }
        ~Stack ( )
        {    delete [ ]    stack ; }
```

```
        incline bool    Add ( const Type item )
      // Push an element onto the stack ; return true
       //    if successful    else return false.
     { if ( StackFull ( ) ) {
        cout << “Stack is full” ; return false ;
        }
        else {
            stack [++top] = item ; return true ;
        }
    }
    incline bool Delete ( Type & item )
      // Pop the top element from the stack ; return true
      // if successful    else return false .
    {    if ( StakEmpty ( ) {
          cout << “Stack is empty” ; return false ;
        }
        else {
           item = stack [top--] ; return true ;
        }
    }
    incline bool StackEmpty ( )
      // Is the stack empty ?
    {    if ( top < 0 ) return true ;
      else return false ; }
    incline bool StackFull ( )
      // Is the Stack full ?
     { if ( top >= (MaxSize – 1 )) return true ;
      else return false ; }
}
```

每个 Add 和 Delete 都花费固定的时间，并且与栈中元素个数无关。StackFull ( ) 和 StackEmpty ( ) 都是时间复杂度为 O(1)的函数。

另一种表示栈的方法是使用链接（或指针），节点是数据和指针信息的集合。栈可由两个域组成的节点来表示，这两个域分别称为 data 域和 link 域。每个节点的数据域包含了栈中的一项，而相应的指针域指向包含栈中下一项的节点。例如，在下图中，含有项 A,B,C,D,E

的以上述顺序插入。

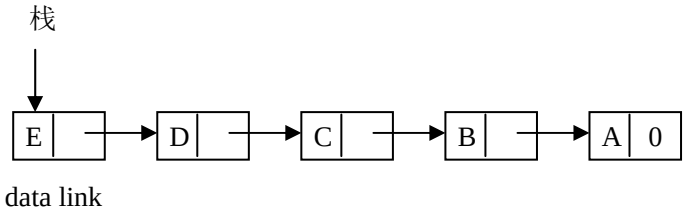


图 2-1-1 含有 5 个元素

程序 2-1-2　　链栈的表示

```
# define NULL    0
# include < iostream.h >
class Stack
{
      private :
          struct node {
              Type data ; struct node    * link ;
          }
      public :
          Stack ( )
          {    top = NULL ; }
          ~Stack ( )
          {    struct node    * temp ;
              while ( top ) {
                  temp = top ;    top = top -> link ;
                  delete temp ;
              }
          }
        bool Add ( const Type item ) ;
        bool Delete ( Type & item ) ;
        inline bool StackEmpty ( )
            // Is the stack empty ?
       {    if ( top ) return false ;
          else return true ;
       }
};
```

top 变量指向表中的最顶层节点（最后插入的那一项）。通过设置 top = 0 表示空栈。程序 2-1-2 是一个类的定义，对应栈的表示。由于采用了指针，所以很容易完成插入和删除操作。例如，要向栈中插入一项应当描述如下

```
bool Stack :: Add (const Type item )
{   struct node * temp = new node ;
     if ( temp ) {   temp -> data = item ; temp -> link = top ;
                  top = temp ; return true ; }
     else { cout << “Out of space!” << endl ;
          return false ;   }
}
```

语句 * temp = new node ; 将一个可用节点的地址赋值给变量 temp 。如果没有其他节点，返回 0。如果存在一个节点，就将适当的值存入该节点的两个域中。然后更新变量 top ，使其指向链表中新的栈顶元素。最后返回 true 。如果空间不足，将打印出错信息并返回 false 。

注意到程序 2-1-2 的类定义中没有 StackFull ( ) ， Add 程序与在程序 2-1-1 中也稍有不同。在链栈中实现 StackFull ( ) 的方法之一是采用如下语句：struct node * temp = new node ; 。对于删除操作，当然完成： 如果 temp 非空，就删除 temp 并返回 true ，否则返回 false 。但是这样会导致插入操作效率不高。为此如下设计删除程序：

```
bool Stack :: Delete ( Type & item )
{   if ( StackEmpty ( ) ) {
        cout << “Stack is empty” << endl ;
        return false ;
    }
    else {
        struct   node   * temp ;
        item = top -> data ; temp = top ;
        top = top -> link ;
        delete   temp ; return true ;
    }
}
```

如果栈为空，删除操作会产生错误信息 Stack is empty 并返回 false 。否则，栈顶元素的值存储到变量 item 中，保持指向第一个节点的指针，更新栈顶 top 指向下一个节点。返回删除的节点以便以后使用，最后返回 true 。

用链表存储比顺序数组 Stack [MaxSize] 更耗空间，然而链表有更大的灵活性，对于许多结构可以同时使用同一个可用的存储池。这两种表示方式的插入和删除操作的次数都与栈的大小无关。

采用数组 q [MaxSize] 可以有效地表示队列，并可以看成是循环的。除了数组，队列类的定义包含了 3 个整型变量：front , rear , MaxSize ，通过增加变量 rear 到下一个空位来插入元素。当 rear = MaxSize – 1 时，如果 q [0] 处有空位，则新元素就插入到该处。变量 front 总是指向队列中第一个元素沿逆时针方向相邻的位置，当且仅当队列为空时，front = rear 成立。初始设置为 front = rear = 0 。图 2.3 描述一个容量 n > 4 并包含从 J1 到 J4 四个元素的循环队列的两种可能的情况。

程序 2-1-3 队列的类定义

```
# define NULL    0
# include < iostream.h >
class queue
{
    private :
       Type * q ;
       int    front , rear , MaxSize ;
    public :
       Queue ( int MSize ) ; MaxSize (MSize )
       { q = new Type [MaxSize ] ; rear = front =0 ; }
       ~Queue ( )
       { delete [ ]    q ;}
       bool    AddQ (Type item ) ;
       bool    DeleteQ ( Type & item ) ;
       bool    QFull ( ) ;
       bool    QEmpty ( ) ;
}
```

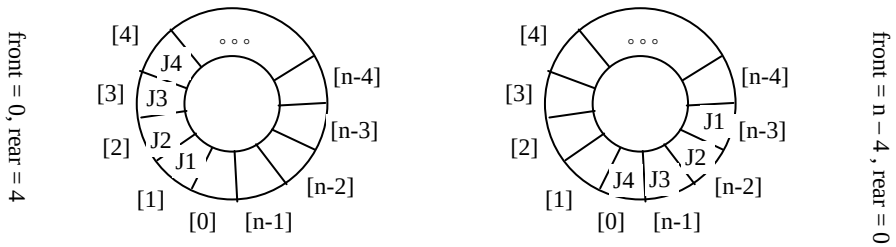


图 2-1-2 含有 4 各元素 J1、J2、J3、J4 的循环队列

要插入一个元素，需要顺时针移动 rear 一个位置。这可以使用下面代码实现：

```
if ( rear = = MaxSize – 1 )    rear = 0 ;
else rear ++ ;
```

但实现该操作的更好的方法是使用内置的计算余数的操作% 。在插入之前，通过 rear = (++ rear) % MaxSize ; 来递增 rear 指针。类似地，每次删除时，要顺时针 移动 front 一个位置。程序 2-1-4a 和程序 2-1-4b 表明：通过把队列看作循环数组，队列的插入和删除操作能在固定的时间（O(1)）内得以实现。

2.2 树

树 T 是含有一个或多个节点的有限集合，其中有一个是特别指定的节点称为根节点，其余的节点被分割成 $n \geq 0$ 个互不相交的集合 $T_1, \cdots, T_n$，这里的每个集合都是一棵树，这些集合称为树 T 的子树。没有子树的节点称为叶节点，既不是根节点又不是叶节点的节点称为内节点。如果节点 X 不是叶节点，则 X 的子树的根节点称为 X 的子节点（或孩子），X 自然就称为这些子节点的父节点（或双亲）。叶节点的度定义为零，其它节点的度为其子节点的个数。从根节点到节点 X 的路径上的所有节点统称为 X 的祖先。

节点的层（层次）递归地定义为：根节点为第 1 层；如果节点 X 位于第 p 层，则它的子节点的层次为 $p+1$ 层。各个节点层次的最大值定义为该树的高度（或深度）。如下图

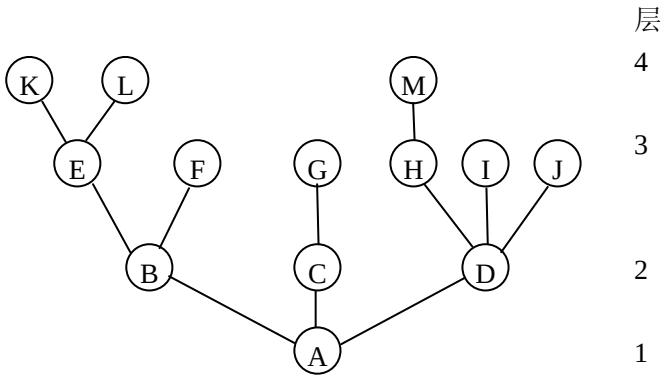
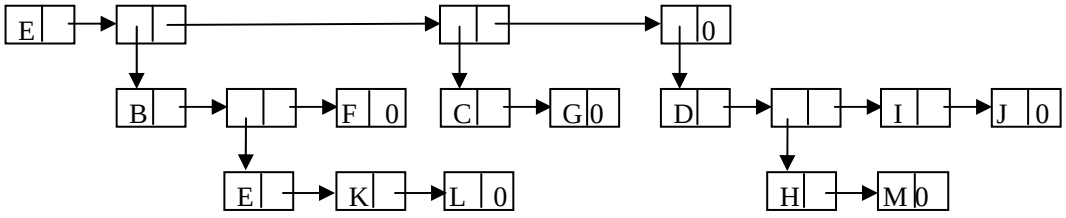


图 2-2-1 树的例

上图很像一棵树，但一般情况下树的示意图都是倒着画的。$n \geq 0$ 个不相交的树的集合称为森林，所以森林可以是空的。将一棵至少有 2 个子树的树的根删掉，则会得到一个森林。

如何在计算机内存中表示一棵树呢？若用链表来表示，其中链表中的每个节点与树的每个节点相对应。由于各个节点的子节点数目不同，每个节点将有不同数目的域。然而在编写算法时，数据中节点的大小固定常常是比较简单的。所以，通常选用节点大小固定的列表结构来表示一棵树。下图给出了图 2-2-1 中树的一种列链 tag = 1 时，data 域不再保存数据项，而是一个指向列表的指针。通过将根存入第一个节点，并将随后的节点指向子列表（每个子列表包含根的一棵子树）来表示一棵树。



节点的 tag 域为 1，表明该节点有向下的指针；否则 tag 域为 0

图 2-2-2 图 2-2-1 中树的列表表示

二叉树是一种特殊的树，每个节点的子节点至多有两个。但允许零个节点情况存在。

定义 2.2.1 二叉树是一个有限的节点集合，该集合要么为空，要么包含一个根节点和两棵不相交的二叉树，这两个二叉树分别称为左子树和右子树。

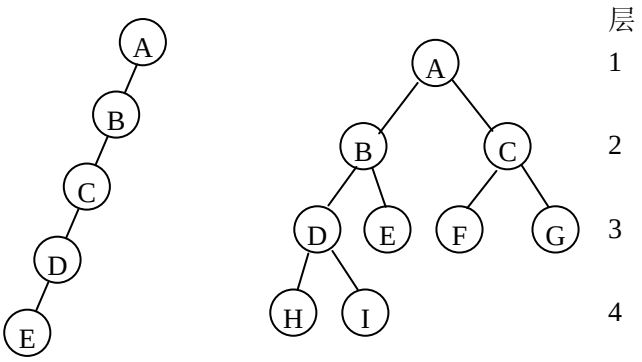


图 2-2-3 二叉树的例

直接计算可知，二叉树的第 i 层节点的数目至多为 2^{i-1}，而高度为 k 的二叉树的节点个数不超过 2^k-1，如果等于 2^k-1，则说这个二叉树为满二叉树。二叉树节点的编号是从第 1 层到第 2 层，等等，每一层都从左子节点开始。根节点编号为 1，然后每数一个加 1，这样最后一个叶节点的编号是 2^k-1。具有 n 个节点高度为 k 的二叉树，当且仅当它的节点编号与高度为 k 的满二叉树的前 n 个节点编号一致时，才说该二叉树是完全二叉树。具有 n 个节点的完全二叉树可以被压缩存储在一维数组 tree[n+1]，其中，编号为 i 的节点存在 tree[i] 中。这种存储可以简单地确定每个节点的父节点、左子节点、右子节点，而不用任何链接信息。

引理 2.2.1 如果按照上述顺序表示具有 n 个节点的完全二叉树，则对于索引为 i 的节点有下列性质：

i. 如果 $i\neq 1$，则 i 的父亲 $\text{parent}(i)$ 位于 $\lfloor i/2 \rfloor$；如果 $i=1$，该节点为根节点，无父亲。

ii. 如果 $2i\le n$，则左子节点 $\text{lchild}(i)$ 位于 $2i$；如果 $2i>n$，则该节点没有左子节点。

iii. 如果 $2i+1\le n$，则右子节点 $\text{rchild}(i)$ 位于 $2i+1$；如果 $2i+1>n$，则该节点没有右子节点。

一般二叉树也可以这种表示，但是浪费空间很大，如高度为 k 的右斜二叉树，在最坏情况下，需要 2^k-1 个位置，却只使用了其中的 k 个。此外，这种表示法具有顺序表示法的普遍缺点，当插入和删除节点时，为了反映其余节点层次的变化，需要移动表中许多节点。通过链表表示很容易克服这个缺点。每一个节点有三个域：lchild、data、rchild。尽管这种结构使确定父节点变得困难，但对于大多数应用还是合适的。对于需要频繁确定父节点这一情

况，可以包含第 4 个域：parent。

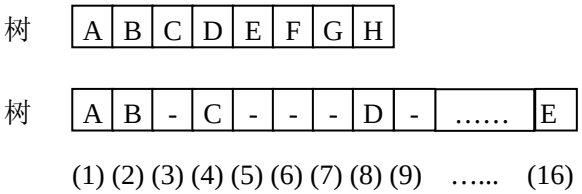


图 2-2-4 树的顺序表示

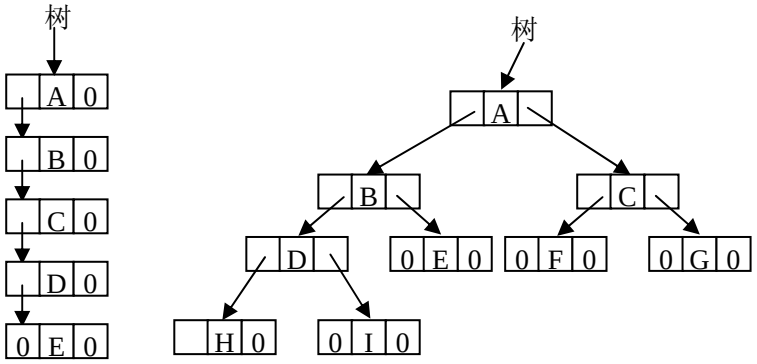


图 2-2-5 图 2-2-4 中二叉树的链表表示

2.3 二叉查找树

定义 2.3.1 二插查找树是一棵二叉树，可以为空二插树，如果不为空，则满足下列要求：

i. 根节点有一个关键字；

ii. 若其左子树不空，则左子树上所有节点的关键字均小于其根节点的关键字；

iii. 若其右子树非空，则右子树上所有节点的关键字均大于根节点的关键字；

iv. 其左右子树也分别为二插查找树。

二插查找树支持查找、插入和删除操作。事实上，可以通过关键字和顺序号来检索数据节点（如检索关键字为 x 的节点，检索第 k 小节点，删除关键字为 x 的节点，删除第 k 小节点，插入一个节点并确定其顺序号，等等）。

程序 2-3-1 二插查找树的类定义

```
class TreeNode{
    friend class BSTree;
    private:
      TreeNode * lchild, * rchild ;
      Type data ;
}
class BSTree
{
```

```
    private:
      TreeNode * tree ;
      TreeNode * Search (TreeNode * t , Type x ) ;
    Public:
       BSTree ( ) {tree = NULL ; }
       ~BSTree ( ) {delete tree ; }
       TreeNode * Search (Type x ) ;        // recurvely search for x .
       TreeNode * ISearch ( Type x ) ;    // Iteratively search for x .
       void Insert ( Type x ) ;    // Insert x .
       void Delete (Type x ) ;    // Delete x .
 };
```

2.3.1 二叉查找树的检索

根据二插查找树定义的递归性质，容易设计一个递归的检索方法。假设要检索一个关键字为 *x* 的节点。节点通常是含有几个域的任意结构，key 是其中的一个域。为简化起见，假定节点仅由 Type 类型的 key 构成。下面将交换使用元素与关键字这两个术语。从根节点开始，若树的根为 NULL，则待检索的树没有元素，因此检索失败；否则，将 *x* 同根节点的关键字进行比较，若 *x* 等于根节点的关键字，则检索成功，终止；若 *x* 小于根节点的关键字，则在右子树中不会存在关键字为 *x* 的节点，只需在根节点的左子树中检索 *x*；若 *x* 大于根节点的关键字，则按类似的方法在根节点的右子树中检索。程序 2-3-2 描述了这个递归检索算法。采用链表结构表示查找树，每个节点有 3 个域：左指针域 lchild、右指针域 rchild、数据域 data 。也可以用程序 2-3-3 的 while 循环实现。

程序 2-3-2 二插查找树的递归检索

```
# define NULL    0
TreeNode * BSTree :: Search ( Type x )
{      return Search ( tree , x ) ; }
TreeNode * BSTree :: Search ( TreeNode * t , Type x )
{      if ( t = = NULL ) return 0 ;
       else if ( x = = t -> data ) return t ;
       else if ( x < t -> data ) return ( Search (t -> lchild , x )) ;
       else return ( Search ( t -> rchild , x )) ;
}
```

程序 2-3-3 二插查找树的迭代检索

```
# define NULL 0
```

```
TreeNode * BSTree :: ISearch ( Type x )
{   bool found = false ;
    TreeNode * t = tree ;
    while ( ( t ) && ( !found )) {
        if ( x = = t -> data ) found = true ;
        else if ( x < t -> data )    t = t -> lchild ;
        else    t = t -> rchild ;
    }
    if ( found ) return t ;
    else return NULL ;
}
```

如果二插查找树的高度为 h,则上述检索的时间复杂度为 $O(h)$。如果根据顺序号检索，每个节点应该有一个附加的域 leftsize，其值等于该节点左子树中节点的个数加 1。算法 2-3-4 给出检索第 k 节点的过程。

程序 2-3-4 二插查找树的顺序号查找

```
# define NULL    0
class TreeNode {
    friend class BSTree ;
    private :
        TreeNode * lchild , * rchild ;
        Type data ; int leftsize ;
};
TreeNode * BSTree :: Searchk ( int k )
{   bool found = false ;    TreeNode * t = tree ;
    while ( ( t ) && ( !found )) {
        if ( k = = ( t -> leftsize ) ) found = true ;
        else if (t < ( t -> leftsize ) )    t = t -> lchild ;
        else {
            k - = ( t -> leftsize ) ;
            t = t -> rchild ;
        }
    }
    if ( found ) return t ;
```

```
    else return NULL ;
}
```

顺序号插值的时间复杂度也为 $O(h)$ 。

2.3.2 二叉查找树的插入

要插入一个新的元素 x，首先要验证 x 的关键字不同于那些已存在节点的关键字。实现这一点要进行检索操作。若检索失败，则在检索终止处插入该元素。如在图 2-3-1a 所示的二插查找树中插入关键字为 80 的节点，首先检索关键字为 80 的节点，检索过程失败，最后一个检查的节点的关键字为 40 的节点，插入新节点使其称为该节点的右子节点，结果如图 2-3-1b 所示。图 2-3-1c 描述了在图 2-3-1b 所示的二插查找树中插入关键字为 35 的元素后的结果。

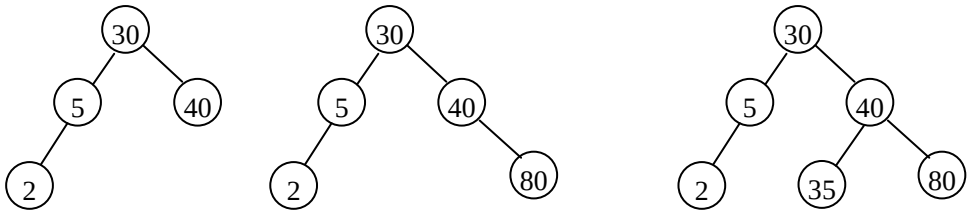


图 2-3-1 二插查找树的插入

程序 2-3-5 给出了上述插入方法的算法描述。若节点含有 leftsize 域，则需要更新该数组。

程序 2-3-5 二插查找树的插入

```
# define NULL    0
void BSTree :: Insert ( Type x )
   //insert x into the binary search tree.
{    bool found = false ;
     TreeNode * p = tree , * q ;
     //Serch for x . q is parent of p .
     while ( ( p ) && ( !found ) ) {
          q = p ; // Save p .
          if ( x = = ( p -> data) )    found = true ;
          else if (x < ( p -> data ) )    p = (p -> lchild ) ;
          else { p = ( p -> rchild ) ;
       }
        // perform insertion .
```

```
    if ( !found ) {
    p = new TreeNode ;
    p -> lchild = NULL ; p -> rchild = NULL ; p -> data = x ;
    if ( tree ) {
       if ( x < ( q -> data ) )    q -> lchild = p ;
       else q -> rchild = p ;
    }
    else    tree = p ;
}
```

插入操作所需要的时间为$O(h)$。

2.3.3 二叉查找树的的删除

在二插查找树中删除一个叶子节点很容易。如从图 2.3.1c)所示的树中删除关键字为 35 的节点，只需要将其父节点的左子域设为 NULL，该节点就被删除了；删除只有一个子节点的非叶子节点也很容易，只需删除包含该元素的节点本身，同时将该节点的子节点上移，取代其位置。如，从图 2.3.1b)所示的树中删除关键字为 5 的节点，只需修改其父节点（关键字为 30 的节点）的指针，使其指向单子节点（关键字为 2 的节点）即可；删除有两个孩子的非叶子节点，可取其左子树中具有最大关键字的节点或取其右子树中具有最小关键字的节点替代要被删除的节点，然后从子树中的相应位置处删除被取节点。如，希望从图 2.3.1c)所示的树中删除关键字为 30 的节点，只需以其左子树中关键字（为 5）最大的节点或以右子树中关键字（为 35）最小的节点取代欲被删除的节点，然后删除被选中的那一个节点。比如，若选择关键字为 35 的节点，则首先将欲被删除的节点的数据修改为 35，然后将关键字为 35 的节点的父节点的左指针设为 NULL 即可。可见，在高度为h二插查找树中执行删除操作的时间为$O(h)$。

2.4 优先队列

提供检索最小（或最大）、插入和删除最小（或最大）元素操作的任何数据结构均称为优先队列。表示优先队列最简单的方法是采用无序线性表。假设队列中有n个元素，并且该队列支持删除最大元素的操作。用顺序结构表示的线性表在表尾插入一个元素很容易，插入操作所需时间为$\Theta(1)$。如果要删除，需要检索关键字最大的元素，因为在无序队列中查找这样的元素所需时间为$\Theta(n)$，所以删除操作所需时间为$\Theta(n)$。还可以采用有序线性表来表示一个优先队列，假设队列中每个元素按非降序排列，则删除操作所需时间为$\Theta(1)$，而插入操作所需时间为$\Theta(n)$。一种均衡删除和插入操作的有序队列结构是堆，若采用最大堆，

则插入和删除操作所需时间均为 $\Theta(\log n)$ 。

2.4.1 堆的基本操作

定义 2.4.1 最大（最小）堆是完全二叉树，具有这样的性质：树中所有非叶子节点的值均不小于（或不大于）其子节点（如果存在的话）的值。

在最大堆中，最大元素在树的根部。可以用数组表示最大堆。向最大堆中插入一个关键值为 x 的元素，是从堆的底部向根检索插入位置，并使得插入节点后仍为完全二叉树，且具有最大堆的性质（程序 2-4-2）；从最大堆中删除一个节点是删除根节点，但是剩下的两棵子树要整合成新的最大堆，整合的方法是：取下底部节点 x，然后从原树的根部向底部检索插入的位置，实现的方法是：将数组中的最后一个元素复制到（原堆的）根，然后调用 Adjust(array,1,Nel)，写在程序 2-4-3 中。下面的算法给出了最大堆的定义、插入操作及删除操作。

程序 2-4-1 最大堆的类定义

```
class Heap
{
    private :
        Type * array ;
        Int MaxSize , Nel ;
            // Max , size of the heap , no. of elements
        void Adjust ( Type a [ ] , int i , n ) ;
    public :
        Heap ( int MSize ) : MaxSize ( MSize )
        { array = new Type [ MaxSize + 1 ] ; Nel = 0 ; }
        ~Heap ( ) { delete [ ] array ; }
        bool Insert ( Type item ) ;
            // insert item
        bool DelMax ( Type & item ) ;
            // delete the maximum .
}
```

程序 2-4-2 最大堆的插入

```
bool Heap :: Insert ( Type item )
{   // Insert item
    int i = ++ Nel ;
    if ( i = = MaxSize ) { cout << “heap size exceeded”   << endl ;
        return false ;
```

```
    }
    while ( ( i > 1) && ( array [ i/2 ] < item ) ) {
        array [ i ] = array [ i/2 ] ; i / = 2 ;
     }
     array [ i ] = item ;    return true ;
}
```

最大堆插入操作的复杂度为 $O(\log n)$

例释：图 2-4-1a) 给出一个具有 5 个元素的最大堆。由于堆是完全二叉树，当加入一个元素形成 6 元素的堆时，其结构形如图 2-4-1b) 所示。如果被插的元素值为 1，则插入后该元素成为 2 的左子节点；若被插元素的值为 5，则该元素不能成为 2 的左子节点，应该把 2 下移为左子节点，如图 2-4-1c) 所示，同时还要决定在最大堆中，5 是否该占据 2 原来的位置。由于父节点 20 大于新插入元素 5，因此 5 可以插在 2 原来所在的位置；又若被插元素的值为 21，则如同图 2-4-1c)一样，把 2 下移为左子节点，由于 21 比原 2 位置的父节点的值还大，所以 21 不能占据原 2 的位置，需要把 20 移到其右子节点，21 插入堆的根节点，如图 2-4-1d) 所示。

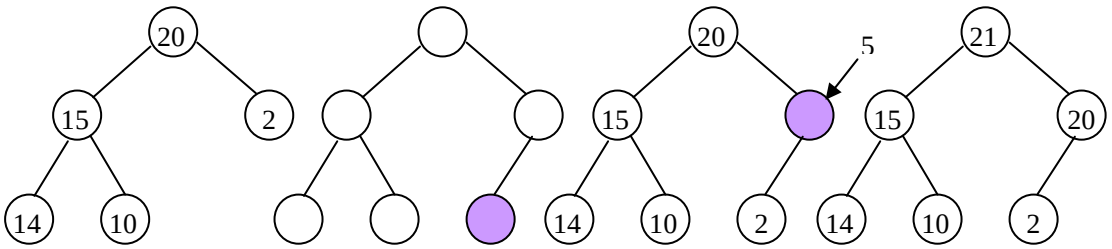


图 2-4-1 最大堆的插入

程序 2-4-3 最大堆的删除

```
void Heap :: Adjust ( Type a [ ] , int i , int n )
   // the complete binary trees with roots 2*i and 2*i + 1 are combined
   // with node i to form a heap rooted at i . No node has an address
   // greater than n or less than 1 .
{
    int j = 2*i ,   item = a [ i ] ;
    while ( j <= n) {
        if ( ( j < n ) && ( a [ j ] < a [ j+1 ] ) )   j ++ ;
           // compare left and right child and let j be the larger child
```

```
            if ( item >= a [j ] ) break ;
                // A position for item is found
            a [ j/2 ] = a [j ] ;    j * = 2 ;
        }
        a [ j/2 ] = item;
    }
    bool Heap :: DelMax ( Type & item )
    {   if ( !Nel ) { cout << "heap is empty" << endl ;
            return false ;
        }
        item = array [1] ; array [1] = array [ Nel -- ] ;
        Adjust ( array , 1 , Nel ) ;    return true ;
    }
```

Adjust 操作的时间复杂度为 $O(\log n)$ 。

例释：从图 2-4-1d)所示的最大堆中删除一个元素，首先将 21 从根部移出，剩下的 5 个元素分布在两个子堆中，需要将这两个子堆重新构造成一个最大堆。为此可以移动位置 6 中的元素，即 2，这样就得到了正确的结构，如图 2-4-2a)所示。但此时根节点为空且元素 2 也不在堆中。如果将 2 直接插入根节点，得到的二叉树不是最大堆，所以，根节点的值应该取 2、根的左子节点的值、根的右子节点的值三数之最大者。这个值为 20，将其移到根节点，此时位置 3 形成一个空位。由于这个位置没有子节点，所以 2 可以直接插入，结果如图 2-4-2b)所示。

如果从图 2-4-2b)所示的最大堆中删除一个元素，则首先将 20 从根部删除，删除之后，堆的二叉树结构应该如图 2-4-2c) 所示。为得到这个结构，10 从位置 5 移出。如果将 10 放在根节点，结果并不是最大堆，此时需将根节点的两个子节点中最大的值，即 15，移到根节点，位置 2 出现空位。如果将 10 直接插入位置 2，则形成的树不是最大堆，因而将 14 上移，将 10 插入位置 4，结果如图 2-4-2d) 。

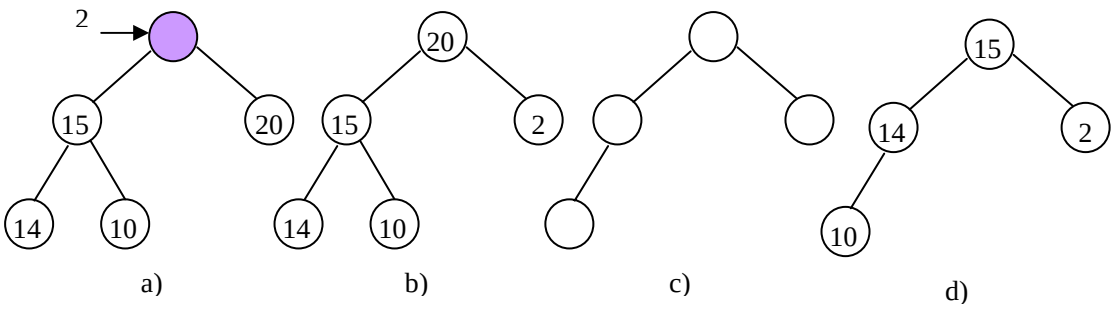


图 2-4-2 最大堆的删除

2.4.2 堆的初始化

给定 n 个元素的数组，可以用 Insert 建立堆，这要从空堆开始，通过逐渐插入元素的办法来完成。实际上，这相当于对已知数组元素进行排列。也可以通过使用类似于 Adjust 的函数 AdjustH 来建立堆，这时我们需要将数组看成一棵完全二叉树，然后在树上从底部到根部逐步进行调整。这两种创建堆的方法分别在程序 2-4-4 和程序 2-4-5 中实现。

程序 2-4-4 逐步插入建立一个最大堆

```
void Sort (Type a [ ] , int n )
  // Sort the elements a [1: n ] .
{   Heap heap ( SIZE ) ;
    for ( int i= 1 ; i <= n ; i+ + )
       heap . Insert ( a [i ] ; Type    x );
    for (i = 1 ; i <= n ; i + + ) {
       heap . DelMax (x ) ;    a [ n- i + 1 ] = x ;
    }
}
```

例释：图 2-4-3 给出了采用 Insert 程序将数据序列(40,80,35,90,45,50,70)建成堆的过程。→左边的树表示调用 Insert 之前数组 heap.array[1:i]的状态，→右边的树表示调用 Insert 之后数组 heap.array[1:i]的状态。整个过程中，数组都被表示成完全二叉树。

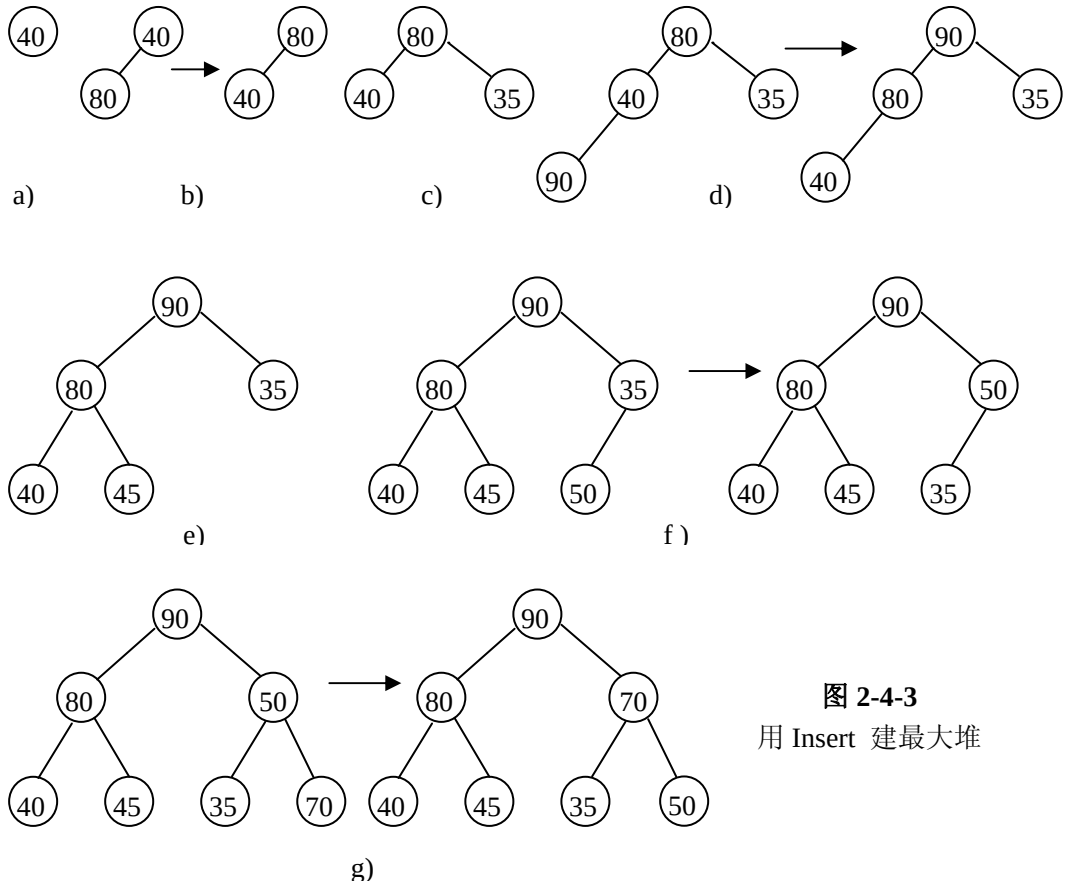


图 2-4-3
用 Insert 建最大堆

在最坏情况下，数据数组的元素按递增排列，每个新元素将会成为新的根节点。在完成二叉树的第i层上，至多有2^{i-1}个节点，$1 \le i \le \lceil \log_2(n+1) \rceil$。第$i$层上的节点到根的距离为$i-1$，因此，使用 Insert 创建堆在最坏情况下所需时间为

$$\sum_{1 \le i \le \lceil \log_2(n+1) \rceil} (i-1)2^{i-1} < \log_2(n+1)2^{\lceil \log_2(n+1) \rceil} = O(n \log n)$$

程序 2-4-5 创建n个任意元素的堆

```
void Heapify (Type a [ ] , int n )
   // Readjust the elements in a [1: n ] to form a heap .
{
    for ( int i = n/2 ; 1 ; i- - )   AdjustH (a , i , n ) ;
}
```

例释：图 2-4-4 给出用 AdjustH 构建最大堆的过程，这一过程将图 2-4-4a)所示的具有$n=10$个节点的完全二叉树转化成最大堆。从已知完全二叉树最后一个具有子节点的节点（即值为 10 的节点）开始，这个节点一定是最后一个节点的父节点，其位置是$i = \lfloor 10/2 \rfloor$。如果以这个元素为根的子树已是最大堆，则此时不需调整，否则必须调整子树使之成为最大堆。随后，继续检查以$i-1, i-2$等节点为根的子树，直到检查到整个二叉树的根节点（其位置为 1）。

最初$i=5$，由于$10>1$,以位置i为根的子树已是最大堆。下一步检查根节点在位置 4 的子树。由于$15<17$，不是最大堆。为将其调整为最大堆，可将 15 与 17 进行交换，得到的树如图 2-4-4b)所示。然后检查以位置 3 为根的子树，为使其变为最大堆，将 80 与 35 进行交换。之后，检查根位于位置 2 的子树，通过重建过程该子树变为最大堆。将该子树重构为最大堆时需确定子节点中值较大的一个，因为$12<17$，所以 17 成为重构子树的根，下一步将 12 与位置 4 的两个子节点中两个值较大的一个进行比较，由于$12<15$，15 被移到位置 4，空位 8 没有子节点，将 12 插入位置 8，形成的二叉树如图 2-4-4c)。最后，检查位置 1，这时以位置 2 或位置 3 为根的子树已是最大堆，然而$20<(\max\{17,80\})$，因此，80 成为最大树的根，当 80 移入根，位置 3 空出。由于$20<(\max\{35,30\})$，位置 3 被 35 占据，最后 20 占据位置 6。图 2-4-4d)显示了最终形成的最大堆。

令$2^{k-1} \le n < 2^k$，其中$k = \lceil \log_2(n+1) \rceil$。对于第$i$层的节点，在最坏情况下，AdjustH 的迭代次数为$k-i$。函数 Heapify 总共所需要的时间与下面的公式成正比：

$$\sum_{1 \le i \le k} (k-i)2^{i-1} = \sum_{1 \le i \le k-1} i2^{k-i-1} \le n \sum_{1 \le i \le k} i/2^i \le 2n = O(n)$$

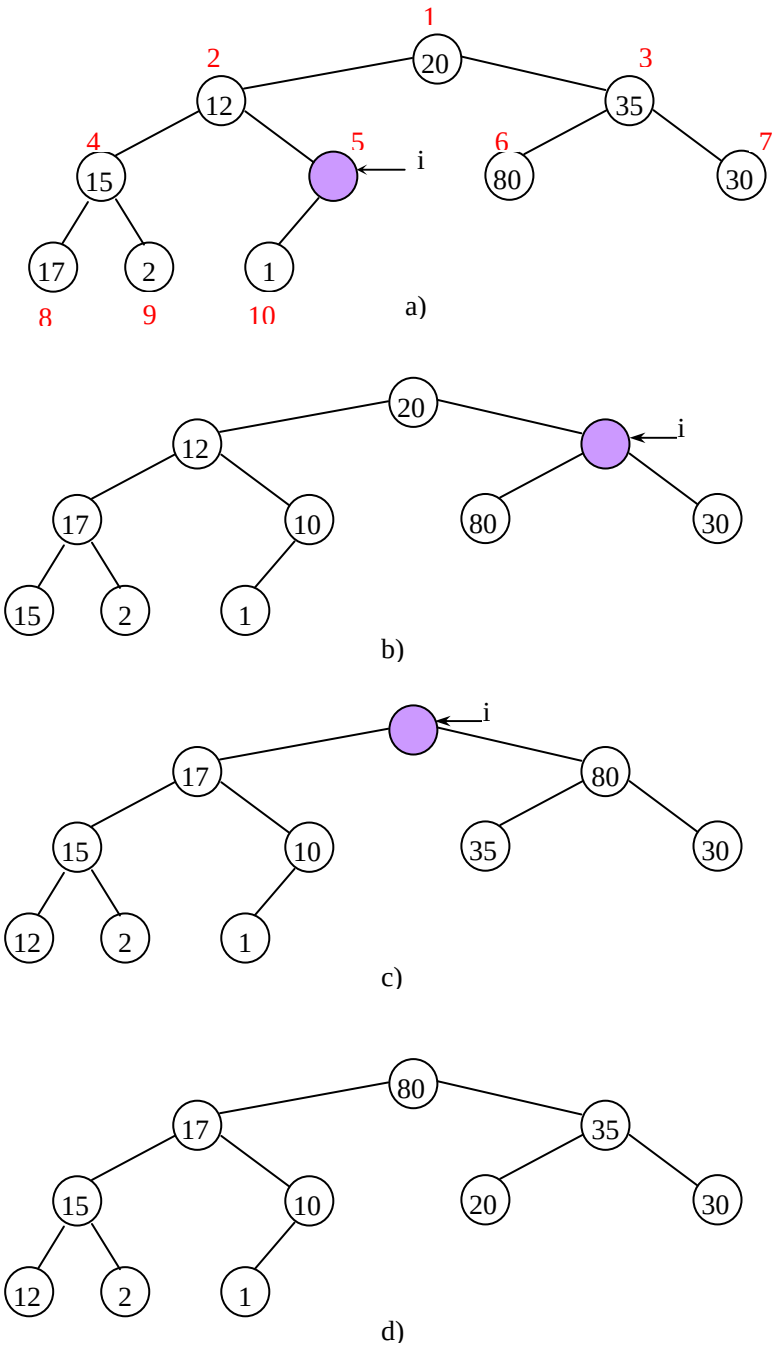


图 2-4-4 用 AdjustH 构建最大堆

2.5 集合的表示和运算

我们打算用树表示一个集合。比如，三个集合 $S_1=\{1,7,8,9\}, S_2=\{2,510\}, S_3=\{3,4,6\}$ 可以用图 2-5-1 中三棵树分别表示出来，当然，表示一个集合的树结构可能不是唯一的，这里的关键是能够判断一个元素是不是在给定的集合中，其中一种方法是将树中非根节点的原来指向其子节点的指针域换成一个指向其父节点的指针域，对于根节点，其父节点可以代表

集合的名，其值规定为 -1。这样，要检查节点 i 是否在已知的树 T 上，只需从节点 i 开始查找它的父节点 j，再查找 j 的父节点 k，直至查到节点 l，而节点 l 的父节点值为 -1 。此时就知道节点 i 是以节点 l 为根的树中的节点。这个过程称为查找，查找的结果记做 $\mathrm{Find}(i)$。如果事先已将某数组元素分成互不相交的集合，比如前面提到的三个集合 S_1, S_2, S_3，而每个集合都已经有树表示，则查找过程能够确定被查找元素属于那个子集合。

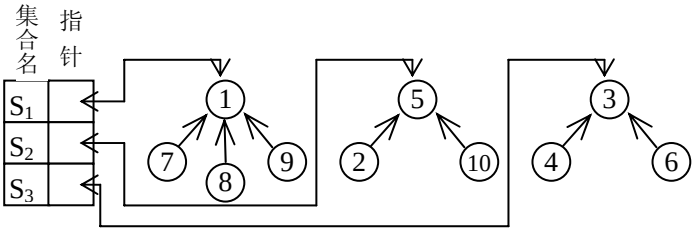


图 2-5-1 S_1、S_2、S_3的数据表示

上述三个子集合的树结构可以用各节点的指针组成的数组表示出来，图 2-5-2 给出了指针与元素的对应关系。

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

图 2-5-2 S_1、S_2、S_3的数组表示

当采用树表示后，合并两个不相交子集就非常简单，只需将代表一棵树的根节点指针修改成指向代表另一棵树的根节点即可。用这种方式合并两个根节点分别为 i, j 的子树形成的一棵更大的树，结果记做 $\mathrm{Union}(i, j)$。程序 2-5-1 给出了以上讨论的并和查找操作的算法。

程序 2-5-1 并和查找的简单算法

```
class Sets
{
    private :
      int * p , n ;
    public :
      Sets ( int Size ) : n (Size )
      {   p = new int [ n +1 ] ;
          for ( int i = 0 ; i <= n ; i + + )
                p [i] = -1 ;
       }
```

```
    ~Sets ( ) { delete [ ] p ; }
    void SimpleUnion ( int i , int j ) ;
    int SimpleFind ( int i ) ;
}
void Sets :: SimpleUnion ( int i , int j )
{   p [i] = j ; }
int Sets :: SimpleFind ( int i )
{ while ( p[i] >= 0 )   i = p [i] ;
  return i ;
}
```

算法 2-5-1 也定义了类 Sets。虽然描述这两个算法很容易，但它们的性能不理想。例如，从 q 个单点集开始，即初始状态由含 q 个单点树构成的森林， $p[i]=-1, 1\le i\le q$ ，依次进行下列并和查找操作：

$$\text{Union}(1,2);\text{Find}(1),\text{Union}(2,3);\text{Find}(2),\text{Union}(3,4);\cdots;\text{Find}(q-2),\text{Union}(q-1,q)$$

则所得到的表示原 q 个单点的集合的树是一棵单枝的树。这里， $\text{Union}(i, j)$ 采用了规则：修改节点 i 的指针为指向节点 j 。如果将此规则稍做修正，变为“向节点多的子树靠拢”，则不会出现上述情况。为此需在每棵树的根节点标出以其为根的树的节点个数 count，由于所有除根节点之外的节点的 p 域都是正数，因此可以在根的 p 域中以负数形式保留 count 值。

程序 2-5-2 使用带权规则的并算法

```
void Sets :: WeightUnion (int i , int j )
    // Union sets with roots i and j , i != j , using the weighting rule .
    // p[i] = = - count[i] , p[j] = = - count[j] .
{
    int temp = p[i] + p[j] ;
    if ( p[i] > p[j] ) { i has fewer nodes .
        p[i] = j ; p[j] = temp ;
    }
    else { // j has fewer or equal nodes .
        p[j] = i ; p[i] = temp ;
    }
}
```

例子 数组 $A=(1,2,3,5,6,7,8)$，考虑从初始状态 $p[i]=-\text{count}[i]=-1, 1\le i\le 8=n$ 开始，依次执行下列并操作的过程：

$$\text{Union}(1,2),\text{Union}(3,4),\text{Union}(5,6),\text{Union}(7,8);\text{Union}(1,3),\text{Union}(5,7);\text{Union}(1,5)$$

则结果如图 2-5-3 表示。

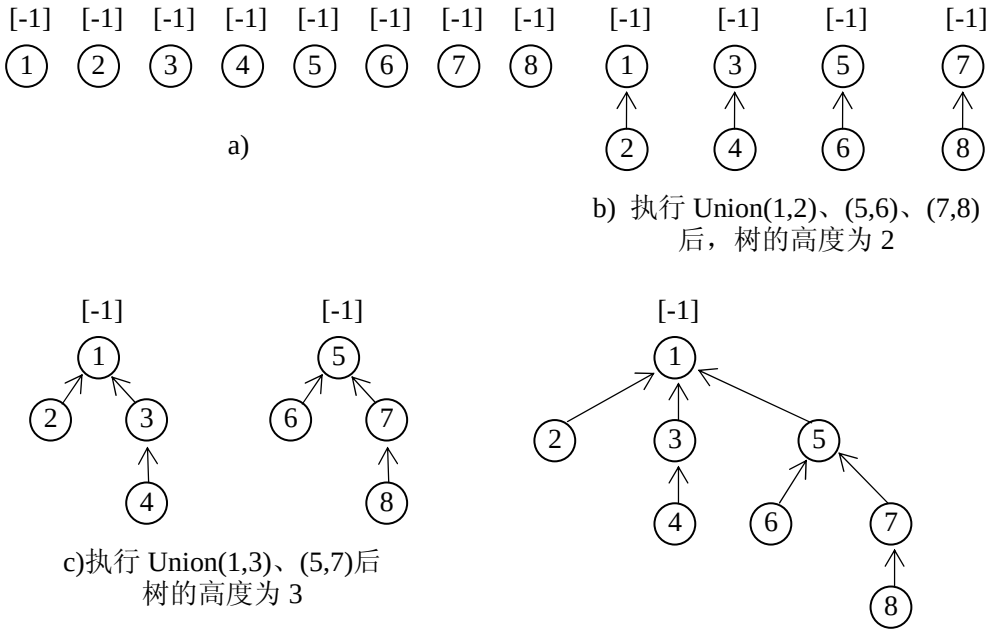


图 2-5-3 采用加权规则处理已知并序列产生的树

命题 2.5.1 假定以森林开始，森林中每一棵树有一个节点。令 T 是由一些并操作生成的具有 m 个节点的树，其中每个并的执行使用 WeightUnion 算法，则 T 的高度不超过 $\lfloor \log_2 m \rfloor+1$。

证明 当 $m=1$ 时，命题成立显然。假定对于有 $i(i\le m-1)$ 个节点的所有上述方法生成的树来说，命题都已成立，以下证明 $i=m$ 时，命题仍成立。令 T 是 WeightUnion 生成的具有 m 个节点的树。考虑最后的并操作 $\text{Union}(k, j)$。令 n 是树 j 中节点的个数，则树 k 中节点的个数为 $m-n$。不失一般性，假定 $1\le n\le m/2$。那么树 T 的高度或者与树 k 的相同，或者比树 j 的高 1。若是前一种情况，则树 T 的高度 $\le\lfloor \log_2(m-a)\rfloor+1\le\lfloor \log_2 m\rfloor+1$。若是后一种情况，则树 T 的高度 $\le\lfloor \log_2 a\rfloor+2\le\lfloor \log_2 m/2\rfloor+2\le\lfloor \log_2 m\rfloor+1$。 证毕

在图 2-5-3 所示的树中查找节点 8 的根需要移动三次指针，如果同时又要查找节点 7 的根还得需要移动两次指针，假如下一次又要查找节点 8 的根，则又要重新移动三次指针。实

际上，有些操作重复执行了。为避免这种浪费，可以采用压缩规则，使得第一次查找节点 8 的根时，就将由 8 到根节点路上的节点(包括节点 8)的指针修改成根节点标号（即 1）。这种做法实际是改变了原来表示集合 $A=\{1,2,3,4,5,6,7,8\}$ 的树最有利于查找的是如下的结构：

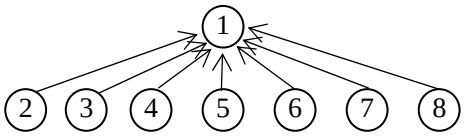


图 2-5-4 采用加权规则生成的最好 8 节点树

程序 2-5-3 采用压缩规则的查找算法

```
int Sets :: CompressFind ( int i )
   // Find the root of the tree containing element i . Use the compressing rule
   // to compress the paths to root for all nodes from i to the root .
{
    int r = i ;
    while ( p[r] > 0 ) r = p[r] ;    // Find root .
    while ( i != r ) { // compress nodes from i to root r .
        int s = p[i] ; p[i] = r ; i = s ;
    }
    return ( r ) ;
}
```

在图 2-5-3d)所示的树中，执行一次 CompreesFind(8)后得到树结构为

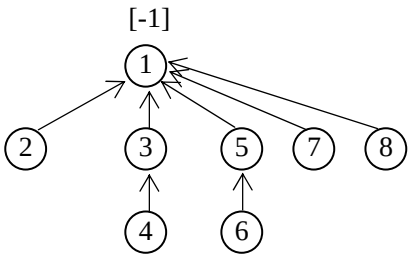


图 2-5-5 执行查找后原树结构发生了变化
更有利于查找操作

命题 2.5.2 假定从 n 个单点树的森林开始，执行了 m 次并和 s 次查找的混合操作序列。记所需的最大时间需求为 $T(s,m)$，则当 $m \geq n/2$ 时，有正常数 k_1, k_2，使得

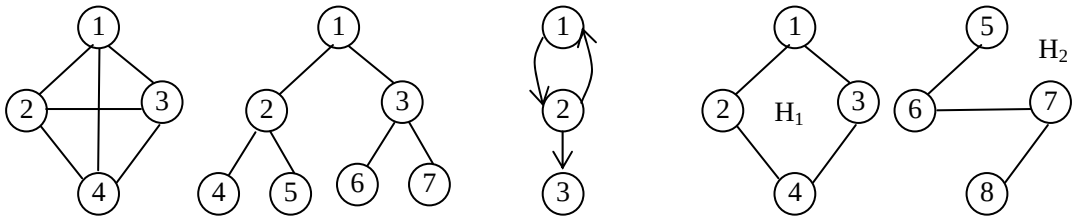
$$k_1\left(n+s\cdot\alpha(s+n,n)\right) \leq T(s,m) \leq k_2\left(n+s\cdot\alpha(s+n,n)\right)$$

其中 $\alpha(p,q)=\min\{i\geq 1 \mid A(i,\lfloor p/q \rfloor)>\log_2 q\}$, $p\geq 1, q\geq 1$，而 $A(i,j)$ 为 Ackermann 数：

$$A(1,j)=2^j, \text{ for } j\geq 1$$
$$A(i,1)=A(i-1,2), \text{ for } i\geq 2$$
$$A(i,j)=A(i-1,A(i,j-1)), \text{ for } i,j\geq 2$$

2.6 图的表示

本节陈述图的三种数据表示方法：邻接矩阵、邻接链表和邻接多重表。始终用 $G=(V,E)$ 代表一个简单图，其节点数 $n\geq 1$。下面是几个图的例子



G_1 G_2 G_3 G_4

图 2-6-1 几个图例

2.6.1 邻接矩阵

G 的邻接矩阵是 $n\times n$ 的 2 维数组，记为 a：当且仅当 (i,j) 是图 G 的一条边时，元素 $a[i][j]=1$，否则 $a[i][j]=0$。根据邻接矩阵很容易知道是否有一条边链接顶点 i 和 j。对于无向图，任何一个顶点的度等于邻接矩阵中对应行的元素之和：$\sum_{1\leq j\leq n} a[i][j]$。对于有向图而言，行元素之和等于对应顶点的出度，而列元素的和等于对应顶点的入度。如果回答图有多少条边？是否连通？等问题，采用邻接矩阵表示时，需要的时间为 $O(n^2)$，因为邻接矩阵除对角线元素外，其余 n^2-n 个元素都要检查。

2.6.2 邻接链表

考虑连通问题时，只需检查图中顶点的链接情况。若图的边不太多，则不必查那些与之不邻接的顶点。采用邻接链表能够有效减少这些不必要的检查。在图的这种表示中，n 行的邻接矩阵被 n 个链表代替。每个顶点都有一个链表，链表 i 上的节点代表了与顶点 i 相邻的顶点。每个节点至少有两个域：顶点和指针。注意每个链表中的节点并不要求按顺序排列。每个链表都有一个头节点。所有的头节点按顺序排列，因而可以很容易的随机访问到邻接链表中任何特定的节点。

程序 2-6-1 邻接链表的类定义

```
class Graph
{
    private :
      int n ; // Number of vertices
      struct node {
          int vertex ;
          struct node * link ;
      }
      struct node * headnodes [n+1] ;
  public :
      Graph ( )
      { for ( int i = 1 ; i <= n; i ++ ) headnodes [i] = NULL ; }
}
```

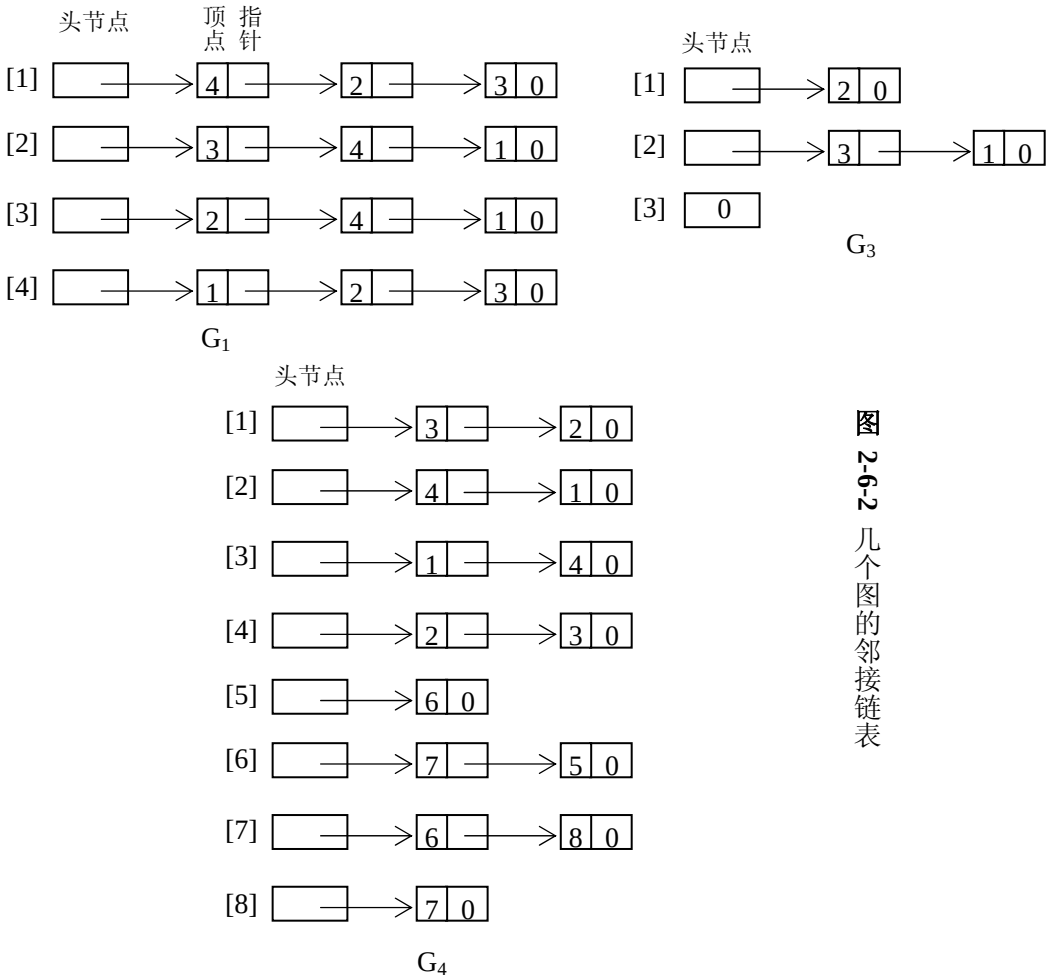


图 2-6-2 几个图的邻接链表

对于有 n 个顶点和 e 条边的无向图，这种表示法需要 n 个头节点和 $2e$ 个表节点。每个表

节点有两个域。由于表示一个值为 m 的数需要占用 $O(\log m)$ 位，所以头节点需要乘以 $\log n$，表节点需要乘以 $\log n+\log e$。通常可以顺序地**压缩存储邻接表**中的节点，因而可以不用指针。这种情况下，可以利用一个数组 $\text{node}[n+2e+2]$。$\text{node}[i]$ 给出了顶点 $i(1\le i\le n)$ 对应的链表在数组中的起始位置。并且 $\text{node}(n+1)$ 被设置为 $n+2e+2$。顶点 i 的邻接点存储在 $\text{node}[i],\cdots,\text{node}[i+1]-1, 1\le i\le n$。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
10	12	14	16	18	19	21	23	24	3	2	4	1	1	4	2	3	6	7	5	6	8	7

图 2-6-3 图G_4的顺序表示

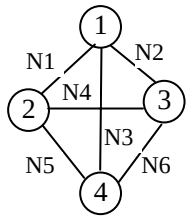
对于有向图而言，其表节点的个数仅仅是 e。任何一个顶点的出度可以由该顶点的邻接表中的表节点个数来确定，因而确定图的有向边的总数需要 $O(n+e)$ 的时间。

2.6.3 邻接多重表

在无向图的邻接表表示中，每条边 (u,v) 由两项来表示，表中的一项来表示 u，而另一项表示 v。在一些应用中需要能够确定某一条指定的边的第二项，并且，当那条边被检查过之后要进行标记。如果将邻接表改造成多重表（即表中的节点可以被多个表共享），那么很容易达到上面提出的要求。多重表仍以顶点作为头节点，而以边作为表节点。表节点由 5 个域组成，结构为

flag	顶点 1	顶点 2	表 1	表 2

其中，flag是占一个位的标志域，可以用来指示这条边是否已经检查过。当我们把所有的边编号以后，所有以 v 为头的边按照编号从小到大的顺序依次链起来，然后挂到顶点 v 上。图 2-6-4b) 给出了图G_1的邻接示意图，其边的编号如图 2-6-4a)，图 2-6-4c)给出了各边对应的



a) G_1

顶点 1：N1→N2→N3
顶点 2：N1→N4→N5
顶点 3：N2→N4→N6
顶点 4：N3→N5→N6

b)

c)

N1		1	2	N2	N4	边 (1,2)
N2		1	3	N3	N4	边 (1,3)
N3		1	4	0	N5	边 (1,4)
N4		2	3	N5	N6	边 (2,3)
N5		2	4	0	N6	边 (2,4)
N6		3	4	0	0	边 (3,4)

图 2-6-4 多重链表

表节点的内容，比如，节点 N1，它代表的边是(1,2)，而 N2、N4 是在链表 b)中链到 N1 的两个表节点。

程序 2-6-4 邻接多重表的定义

```
class Graph
{
    private :
      int n ;    // Number of vertices
      struct edge {
        bool    flag ;
        int vertex1 , vertex 2 ;
        struct edge * path1 , * path2 ;
      }
     struct edge * headnodes [n+1] ;
  public :
     Graph ( )
     { for (int i = 1 ; i <= n ; i++ ) headnodes [i] = NULL ; }
}
```

附录 3. ALGEN 语言

对于算法的描述可以采用一种高级计算机语言，如 C/C++、Java、Fortran、Basic 等，但是这些语言都要强调每一步实现的细节，这对于分析算法的性能不是完全必要的。描述算法最要紧的是表达清楚算法的基本思想和关键过程，选用简明易于理解同时便于人工或采用机器翻译成其它实际使用的计算机高级语言是必要的。为此，我们采用 ALGEN 语言，它与 ALGOL 语言和 PASCAL 语言相似，符合人们通常的表达习惯。

1．变量类型的声明

在 ELGEN 语言中，只作简单的变量类型区别，它们是 integer，real，bool 和 char。这主要考虑到对这些对象进行操作（或运算）的特殊要求。所有操作（或运算）过程都理解为精确的。当需要考虑数值计算的精度时，需在相应的位置加语句 digit:= n；表示保留小数点后 n 位。这个约束的作用从该语句开始直到出现语句 end{digit}时结束。此外，用 evalf(a,n)表示对数值 a 保留 n 位小数；用 floor(a)表示不大于数值 a 的最大整数，ceil(a)表示不小于数值 a 的最小整数。

2．保留字

变量的命名是以字母开头的字母与数字及下划线组成的字, 区分大小写, 但不准采用保留字, 这里的保留字主要包括以下字:

逻辑符: and、or、not、true、false

结束符：exit、return、end、break、continue

语句关键字： if 、elif、 then、 else ；for、from、by、to、do、while、in、loop、until

关系符：<、>、=、≤、≥、≠、……

3．对变量的赋值采用 v:=expr；其中 v 是变量（名），expr 是一个（表达式的）值。

4．for 循环语句

for i **from** s **by** d **to** t **do**

statements;

end{for}

当起始值 s 为 1 时，from s 可以省略；当步长 d 为 1 时，by d 可以省略；

for i **in** S conds **do**

statements;

end{for}

S 可以是一个集合、序列或表，conds 表示约束条件，可根据情况选取或不选取，所以是可选项。

5． while 循环与 loop 循环语句

while conds **do**

statements;

end{while}

loop

statements；

until conds；

end{loop}

6．if 条件语句

if cond0 **then**

statement0；

elif cond1 **then**

statement1；

……..

elif condk **then**

statementk；

else

statements；

end{if}

这里，elif、else 及它们后边的语句都是可选项。条件 cond0、cond1、…、condk 之间都是不相交的.

7．case 条件语句

为使用方便，ALGEN 也使用 case 条件语句，格式如下：

case :

c1: statement1;

…….

ck: statementk;

else statement;

end{case}

其中，else 是选项，而前面的 k 种情况不能为空，且所有情况之间是相互排斥的。

8．进程: proc name(formal parameters)

进程是指独立完成一个任务的程序段。一个进程可以调用其它进程，也可以被其它进程所调用。这里的进程有两种，一是纯过程，它可能会改变输入参数，但没有返回值；另一类有返回值，称为函数。前者的调用方式为 name(actual parameters)；后者的调用方式为 v:=name(actual parameters)；因而，后者必须带有

语句 retrurn(expr); 指出返回的内容。而前者是通过全局变量将信息返回给调用它的进程的。主进程的返回信息被存在内存中。

一个进程的完整格式为：

```
proc name (formal parameters)
    global variables;
    local variables;
    statements;
end{name}
```

9. break 强制结束当前循环体；continue 强制跳过本次循环；exit 强制结束当前进程；return 强制结束当前进程并返回后跟其括号中的内容。end 是任何进程代码结束的标志符。

10. go to label 从当前位置转到带有标号 label 的语句，label 后跟冒号:。这个语句用于跳出 while 循环体以及跳出递归程序体都是比较方便的，但尽量少用，因为它可能会使整个程序的逻辑关系变得晦涩难懂。

11. 数组是所有数据结构的基础，这里用 A[1..n]表示一个长为 n 的一维数组，它的第 i 个元素记为 A[i]，而 A[k..m]表示子块，它是该数组的一个连贯部分：从元素 A[k]到元素 A[m]。二维数组定义为 A[1..m,1..n]，它对应于一个 m×n 矩阵。更高维的数组可类似定义，它们的子块也可类似表示。给数组赋值通过给元素赋值完成。

12. 有时为叙述方便简洁，ALGEN 允许使用数学表达式和自然语言，但含义必须是清楚明确的。

13. 每个执行语句都以 ；号结尾；注释语句的每一行都以双斜杠 // 开始。