草稿

目录

周鹏

第一章　Java 与面向对象程序设计

在这一章中向读者简要介绍有关 Java 的基本知识。Java 语言是一种广泛使用并且具有许多良好的如面向对象、可移植性、健壮性等特性的计算机高级程序设计语言，在这里对 Java 的介绍不可能面面俱到，因此在第一章中只对理解书中 Java 代码的相关知识进行介绍。对于熟悉 Java 的读者可以不阅读本章。

1.1 Java 语言基础知识

1.1.1 基本数据类型及运算

在 Java 中每个变量在使用前均必须声明它的类型。Java 共有八种基本数据类型：四种是整型，两种浮点型，一种字符型以及用于表示真假的布尔类型。各种数据类型的细节如表 1-1 所示。

表 1-1 Java 数据类型

类型	存储空间	范围
int	32 bit	[-2147483648,2147483647]
short	16 bit	[-32768,32767]
long	64 bit	[-9223372036854775808, 9223372036854775807]
byte	8 bit	[-128,127]
float	32 bit	[-3.4E38,3.4E38]
double	64 bit	[-1.7E308,1.7E308]
char	16 bit	Unicode 字符
boolean	1 bit	True,false

在声明一个变量时，应先给出此变量的类型，随后写上变量名。在声明变量时一行中可以有声明多个变量，并且可以在声明变量的同时对变量进行初始化。例如：

```
int i;
double x, y = 1.2;
char c = ‘z’;
boolean flag;
```

在程序设计中，常常需要在不同的数字数据类型之间进行转换。图 1-1 给出了数字类型间的合法转换。

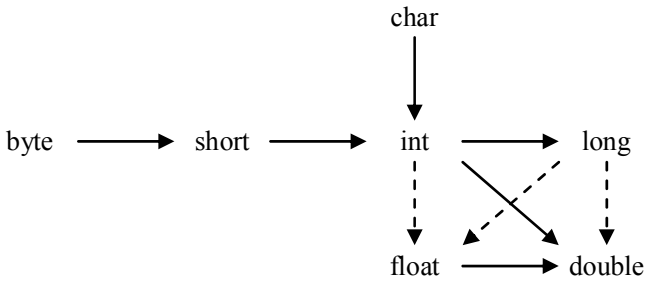


图 1-1 数字类型间的合法转换

图 1-1 中六个实箭头表示无信息损失的转换，而三个虚箭头表示的转换则可能会丢失精度。

有时在程序设计中也需要进行在图 1-1 中没有出现的转换，在 Java 中这种数字转换也是可以进行的，不过信息可能会丢失。在可能丢失信息的情况下进行的转换是通过强制类型转换来完成的。其语法是在需要进行强制类型转换的表达式前使用圆括号，圆括号中是需要转换的目标类型。例如：

```
double x = 7.8;
int n = (int)x;        //x 等于 7
```

Java 使用常见的算术运算符＋－*／进行加、减、乘、除的运算。当除法运算符／作用于两个整数时，是进行整数除法。整数的模（即求余）运算使用 % 运算符。对整型变量一种最常见的操作就是递增与递减运算，与 C/C++ 一样 Java 也支持递增和递减运算符。例如：

```
int n = 7, m = 2;
double d = 7;
n = n / m;             //n 等于 3
d /= m;                //d 等于 3.5
n--;                   //n 等于 2
int a = 2 * n++;       //a 等于 4
int b = 2 * ++m;       //b 等于 6
```

此外 Java 还具有完备的关系运算符，如==（是否相等），＜（小于），＞（大于），<=（小于等于），>=（大于等于），!=（不等于）；并且 Java 使用&&表示逻辑与，||表示逻辑或，!表示逻辑非；以及七种位运算符&（与）、｜（或）、^（异或）、~（非）、 >>（右移）、<<（左移）、>>>（高位填充 0 的右移）。

最后 Java 还支持一种三元运算符 ?: ，这个运算符有时很有用。它的形式为

condition ? e1 : e2

这是一个表达式，在 condition 为 true 时返回值为 e1，否则为 e2。例如：

min = x < y ? x : y;

则 min 为 x 与 y 中的较小值。

1.1.2 流程控制语句

计算机高级语言程序设计中共有三种流程结构，分别是：顺序、分支、循环。其中分支与循环流程结构需要使用固定语法的流程控制语句来完成。

Java 中有两种语句可用于分支结构，一种是 if 条件语句，另一种是 switch 多选择语句。

条件语句的形式如下：

if (condition) statement1 else statement2

当 if 后的条件 condition 的值为 true 时执行 statement1 中的语句，否则执行 statement2 中的语句。

多选择语句的形式为：

```
switch (integer expression) {
    case value1: block1; break;
    case value2: block2; break;
    …
    case valueN: blockN; break;
    default: default block;
}
```

switch 语句从与选择值相匹配的 case 标签处开始执行，一直执行到下一个 break 处或者 switch 的末尾。如果没有相匹配的 case 标签，而且存在 default 子句，那么执行 default 子句。如果没有相匹配的 case 标签，并且没有 default 子句，则结束 switch 语句的执行，执行 switch 后面的语句。

Java 中的循环语句主要有三种，分别是 for 循环、while 循环、do…while 循环。

for 循环的形式为：

for (initialization; condition; increment) statement;

for 语句的循环控制的第一部分通常是对循环变量的初始化，第二部分给出进行循环的测试条件，第三部分则是对循环变量的更新。

while 循环的形式为：

while (condition) statement;

while 循环首先对循环条件进行测试，只有在循环条件满足的情况下才执行循环体。

do…while 循环的形式为：

do statement while (condition);

与 while 循环不同的是，do…while 循环首先执行一次循环体，当循环条件满足时则继续进行下一次循环。

1.1.3 字符串

字符串是指一个字符序列。在 Java 中没有内置的字符串类型，而是在标准 Java 库中包含一个名为 String 的预定义类。每个被一对双引号括起来的字符序列均是 String 类的一个实例。字符串可以使用如下方式定义。

```
String s1 = null;          //s1 指向 NULL
String s2 = "";            //s2 是一个不包含字符的空字符串
String s3 = "Hello";
```

Java 允许使用符号 ＋ 把两个字符串连接在一起。当连接一个字符串和一个非字符串时，后者将被转换成字符串，然后进行连接。例如：

```
s3 = s3 + "World!";        //s3 为"HelloWorld!"
String s4 = "abc" + 123;   //s4 为"abc123"
```

Java 的 String 类包含许多方法，其中多数均非常有用，在表 1-2 中只给出最常用的一些方法。

表 1-2 Java String 类常用方法

char	**charAt**(int index) Returns the char value at the specified index.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
boolean	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(Object anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(String anotherString) Compares this String to another String, ignoring case considerations.
int	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
int	**lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	**length**() Returns the length of this string.
boolean	**startsWith**(String prefix) Tests if this string starts with the specified prefix.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
char[]	**toCharArray**() Converts this string to a new character array.

String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.

如果读者需要进一步了解有关String提供的其他方法及方法完成的功能，可以通过在线API（应用程序接口）文档了解相关信息，从中你可以查到标准库中所有的类及方法。API文档是Java SDK的一部分，以HTML格式显示。JDK1.5.0 的API文档地址为：http://java.sun.com/j2se/1.5.0/docs/api/index.html。

1.1.4 数组

数组是用来存放一组具有相同类型数据的数据结构。可以通过整型下标来访问数组中的每一个值。数组是可以通过在某种数据类型后面加上[]来定义，在此之后跟上变量名就可以定义相应类型的数组变量了。例如：

```
int[] a;
```

还可以使用另一种方法定义数组，例如：

```
int a[];
```

以上这两种方法的定义是等价的。在这里只定义了一个整型数组变量a，但是还没有将a真正的初始化为一个数组。为将一个数组初始化可以使用new关键字，也可以使用赋值语句进行初始化。数组一旦被创建，就不能改变它的大小。

例如：

```
a = new int[10];     //将 a 初始化为大小为 10 的整型数组。
int[] b = {0,1,2,3}  //将 b 初始化为大小为 4 的整型数组，
                     //并且 4 个分量的值分别等于 0，1，2，3
```

数组的下标从0开始计数，到数组大小减1结束。在Java中不能越过数组下标的范围去访问数组中的数据。例如：

```
a[10] = 10;
```

如果越过数组的下标访问数据，则会产生一个名为ArrayIndexOutOfBoundsException的运行时错误。为避免产生这种错误，可以通过在访问某个下标的数组元素前检查数组的大小来避免。数组的大小可以通过数组的变量length返回。例如：

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

由于在Java中数组实际上是一个类，因此两个数组变量可以指向同一个数组。请读者

预测以下这段代码的运行结果：

```
int[] a = {1,1,1};
int[] b = a;
for (int i=0;i<b.length;i++)
      b[i]++;
for (int i=0;i<a.length;i++)
      System.out.print(a[i]);
```

在这段代码中对数组 b 的每个分量加 1，但是在输出数组 a 的每个分量时，可以发现 a 的每个分量都发生了变化，都为 2。其原因是赋值语句 int[] b = a; 只是将对于数组 a 的引用传递给变量 b，此时数组变量 a、b 实际上是指向同一个数组空间。图 1-2 说明了这段代码运行时的情况。

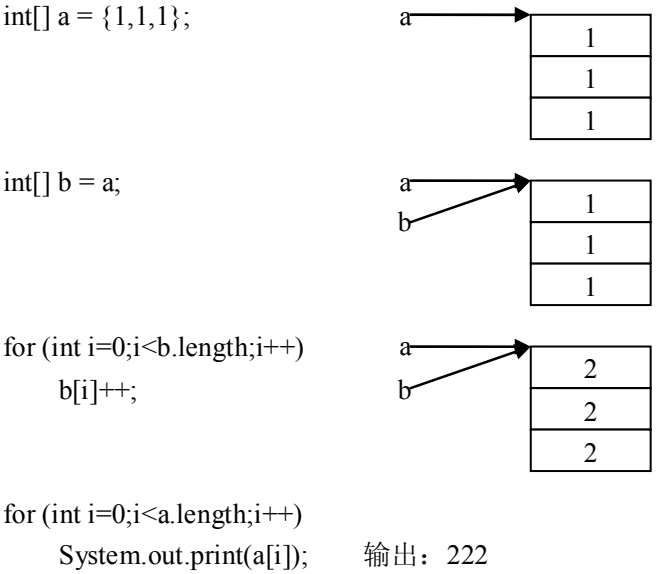


图 1-2 两个数组变量指向同一数组空间

为了得到两个不同的但每个分量的值均相等的数组，可以使用循环语句或 System 类中的 arraycopy 方法来完成。

同样当数组作为方法的参数传递时，也是传递的对于数组的引用，因此在方法中对数组进行的操作会影响到原来的数组。例如：

```
public void changeArray(int[] a){
      for (int i=0;i<a.length;i++)
            a[i] = a[i] * 2;
}
```

那么如下代码

```
int c = {2,2,2};
changeArray( c );
```

```
for (int i=0;i<c.length;i++)
    System.out.print(c[i]);
```

的运行结果为：444。

1.2 Java 的面向对象特性

面向对象程序设计（OOP, Object Oriented Programming）是目前主流的程序设计方法，它已经取代了以前基于过程的程序设计技术。面向对象程序设计主要是指在程序设计中采用抽象、封装、继承等设计方法。

1.2.1 类与对象

在面向对象思想中，抽象决定了对象的对外形象、内部结构以及处理对象的外部接口，其关键是处理对象的可见外部特征。抽象主要是从现实世界中抽象出合理的对象结构。

封装性是保证软件部件具有优良的模块性的基础。在 Java 中，最基本的封装单元是类，一个类定义了将由一组对象所共享的行为（数据和代码）。一个类的每个对象均包含它所定义的结构与行为，这些对象就像是一个模子铸造出来的。所以对象也叫做类的实例。

- **类的定义**

在定义一个类时，需要指定构成该类的代码与数据特别是，类所定义的对象叫做成员变量。操作数据的代码叫做成员方法.方法定义怎样使用成员变量，这意味着类的行为和接口要由操作数据的方法来定义。

由于类的用途是封装复杂性，所以类的内部有隐藏实现复杂性的机制。所以 Java 中提供了私有和公有的访问模式，类的公有接口代表外部的用户应该知道或可以知道的每件东西。私有的方法数据只能通过该类的成员代码来访问。

在 Java 中类的定义是通过关键字 class 来实现的。例如：

```
public class People {
    private String name;
    private String id;
    //Constructor
    public People(){
        this("","");
    }

    public People(String name,String _id){
        this.name = name;
        id = _id;
    }

    public void sayHello(){
        System.out.println("Hello! My name is " + name);
    }
```

```
    public void sayHello(String name){
        System.out.println("Hello, " + name + "! My name is " + this.name);
    }

    //get & set methods
    public void setName(String name){
        this.name = name;
    }
    public void setId(String id){
        this.id = id;
    }
    public String getName(){
        return this.name;
    }
    public String getId(){
        return this.id;
    }
}
```

代码中使用 class 关键字定义了一个名为 People 的类，class 前面的 public 关键字表示这个类似一个公有类，可被访问。People 类中使用了 this 关键字，this 关键字主要有两个作用，一是表示对隐式参数的引用，一是调用类中的其他构造方法。

在类的内部首先通过 private 关键字定义了两个私有的成员变量，由于这两个成员变量是私有的，因此为了使得能够在类的外部获取或修改这些信息，定义了四个 get、set 方法。

并且 People 类定义了两个构造方法，构造方法是一种特殊的方法，其作用是构造并初始化对象，在一个类中构造方法可以定义多个。要构造一个新的对象只需要在构造方法前使用 new 关键字就可以了。例如：

People jack = new People("Jack","0001");

此外在类中还定义了两个 sayHello 方法实现与外界的互操作。

■ 使用现有类

Java 提供了大量的预定义类可供使用，同时程序中可能还会使用第三方提供的或者自己编写的属于其它包的类，使用这些类会给编写程序带来巨大的便利。

如果在某个类中需要使用其它包中的类，可以使用关键字 import 将需要使用的类在类的定义开始之前引入。

例如在 People 类中还可以定义一个新的成员变量 birthday，而日期可以使用 Java 提供的预定义类 Calendar 实现。以下代码给出了实现方法。

```
import java.util.Calendar;
public class People {
    private String name;
    private String id;
    private Calendar birthday;
```

```
    //构造方法
    ……
    //sayHello 方法、get & set 方法
    ……
}//end of class
```

1.2.2 继承

继承是子类自动获取父类的数据和方法的机制，这是类之间的一种关系。在定义和实现一个类的时候，可以在一个已经存在的类的基础之上来进行，把这个已经存在的类所定义的内容作为自己的内容，并加入若干新的内容。

例如学生也是人，那么他（她）也有 name、id、birthday 等属性，也应该可以和外界进行 sayHello 方法定义的互操作，但是学生是一群特定的人群，他们还具有一些特定的属性以及特定的和外界进行互操作的方法。在这种情况下就需要使用继承，可以定义一个新的类 Student，然后向它添加功能。但是新的类可以重用 People 类中已有的成员变量和方法。抽象的说，Student 和 People 是一个明显的“is-a”关系：每个学生都是人。“is-a”关系就是继承的特点。

在 Java 中使用关键字 extends 来实现继承。例如下面的代码定义了一个新的类 Student，它继承了最初定义的 People 类：

```
public class Student extends People{
    private String sId;  //学号
    //Constructor
    public Student() {
        this("","","");
    }
    public Student(String name,String id,String sId){
        super(name,id);
        this.sId = sId;
    }

    public void sayHello(){
        super.sayHello();
        System.out.println("I am a student of department of computer science.");
    }
    //get & set method
    public String getSId(){
        return this.sId;
    }
    public void setSId(String sId){
        this.sId = sId;
    }
}
```

代码中使用了关键字 super，super 关键字主要有两个作用，一是调用父类的构造方法，一是调用父类的方法。

关键字 extends 表明使用它构造出来的类是从一个现有的类衍生出来的。现有类被称为父类，而新的类称为子类。父类与子类相比并不具有更多的属性和功能，事实上恰恰相反，子类比父类具有更多的属性和功能。“is-a”规则表明子类的每个对象都是父类的对象，例如每个学生都是人。因此，无论何时只要在程序中需要一个父类对象时都可以使用一个子类的对象来替代它，反过来则不行。

例如，可以把子类的对象赋给父类变量：

```
People p = new Student("Bob","0002","2006137129");
```

如果想要把对某个类的对象引用转换为对另一个类的对象引用，需要用圆括号把目标类名括起来，然后放到需要转换的对象引用之前。例如：

```
Student s = (Student)p;
```

当然这种转换并不是一定能够完成，如果是不能完成的情况，程序在运行时会抛出异常。为了使转换在允许的情况进行，可以使用 instanceof 关键字。例如：

```
if ( p instanceof Student)
    Student s = (Student)p;
```

在 Java 中有一个非常特殊的预定义类，那就是 Object 类。在 Java 中 Object 类是所有类的祖先，每个类都有 Object 类扩展而来。在定义类时如果不指定父类，则 Java 会自动把 Object 类作为要定义类的父类。例如 People 类就是 Object 类的子类。因此可以使用 Object 类的变量引用任意类型的对象。例如：

```
Object obj = new People("Jack","0001");
```

在 Java 中不支持多继承。Java 对于多继承大部分功能的实现是通过接口机制来完成的。

1.2.3 接口

接口是 Java 实现多继承的一种机制，一个类可以实现一个或多个接口。接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现，因此这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为。简单的说接口不是类，但是定义了一组对类的要求，实现接口的某些类要与接口一致。

在 Java 中使用关键字 interface 来定义接口。例如：

```
public interface Compare {
    public int compare(Object otherObj);
}
```

Compare 接口定义了一种操作 compare，该操作应当完成与另一个对象进行比较的功能。它假定某个实现这一接口的类的对象 x 在调用该方法时，例如 x . compare(y)，如果 x 小于 y，返回负数，相等返回 0，否则返回正数。

让类实现一个接口需要使用关键字 implements，然后在类中实现接口所定义的方法。例如：

```
public class Student extends People implements Compare{
    private String sId;   //学号
    //Constructor
```

```
    public Student() {
        this("","","");
    }
    public Student(String name,String id,String sId){
        super(name,id);
        this.sId = sId;
    }

    public void sayHello(){
        super.sayHello();
        System.out.println("I am a student of department of computer science.");
    }
    //get & set method
    public String getSId(){
        return this.sId;}
    public void setSId(String sId){
        this.sId = sId;}

    //implements Compare interface
    public int compare(Object otherObj){
        Student other = (Student)otherObj;
        return this.sId.compareTo(other.sId);
    }
}//end of class
```

代码中 Student 类实现了 Compare 接口，并且实现了 compare 方法，这里假定通过两个学生学号的字典顺序完成对学生的比较，学号字典顺序在前的学生小于学号字典顺序在后的学生。

需要注意的是在 Java 中接口不是类，因此不能使用 new 实例化接口。但是不过虽然不能通过 new 构造接口对象，但是还是可以声明接口变量。并且只要类实现了接口，就可以在任何需要该接口的地方使用这个接口的对象。例如：

```
Compare com = new Student("Cary","0003","2006137101");
```

反之也可以将一个接口变量转换为对某个类的对象的引用，不过此时要进行强制转换，并且不一定能够完成，情况和从父类引用到子类引用的转换是一样的。例如：

```
Student s = (Student) com;
```

1.3 异常

对于程序运行时碰到的异常情况，Java 使用了一种被称为“异常处理”的机制来进行处理。在 Java 中一个异常对象总是 Throwable 子类的实例。图 1-3 是 Java 异常继承层次结构的图示。

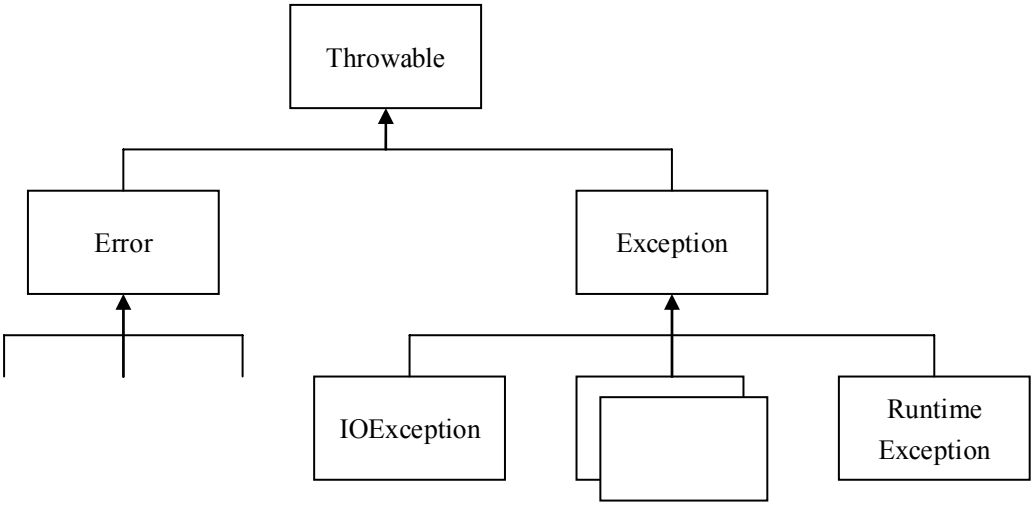


图 1-3 Java 异常层次结构

在 Java 程序设计中，我们关注于 Exception 这个分支体系，而 Exception 中一类是从 RuntimeException 衍生出来的子类，以及不是从它衍生出来的其他异常类。一般来说由编程导致的错误会引起 RuntimeException，例如数组下标越界、错误的类型转换、访问空指针等错误会导致不同类型的 RuntimeException。

在程序中可能会碰到任何标准异常都不能很好描述的异常情况。此时，可创建自己的异常类，创建自己的异常只需要继承 Exception 类或 Exception 的子类就可以了。

在程序中如果碰到了异常的情况，可以有两种方法来处理这个异常，一种是有方法本身捕获这个异常并进行相应的处理，使用 try…catch 结构；另一种是将这个异常从方法中抛出，使用 throws 以及 throw 关键字。例如：

```
public void method1(){
    try{
        //statement may cause exception
        ……
    }catch(ExceptionType e){
        //deal with exception
        ……
    }
}
```

或者例如：

```
Public void method2() throws ExceptionType {
    ……
    if (exception condition) throw instance of ExceptionType;
    ……
}
```

1.4 Java 与指针

尽管在 Java 中没有显式的使用指针并且也不允许程序员使用指针，而实际上对象的访问就是使用指针来实现的。一个对象会从实际的存储空间的某个位置开始占据一定数量的存

储体。该对象的指针就是一个保存了对象的存储地址的变量，并且这个存储地址就是对象在存储空间中的起始地址。在许多高级语言中指针是一种数据类型，而在 Java 中是使用对象的引用来替代的。

考虑前面我们定义的 People 类，以及下列语句：

People p = null; q = new People("Jack","0001");

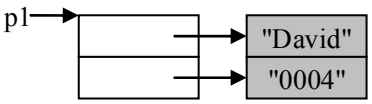
这里创建了两个对于对象引用的变量 p 和 q。变量 p 初始化为 null，null 是一个空指针，它不指向任何地方，也就是它不指向任何类的对象，因此 null 可以赋值给任何类的对象的引用。变量 q 是一个对于 People 类的实例的引用，操作符 new 的作用实际上是为对象开辟足够的内存空间，而引用 p 是指向这一内存空间地址的指针。

为此请读者考虑如下代码的运行结果：

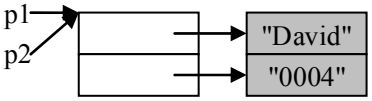
```
People p1 = new People("David","0004");
People p2 = p1;
p2.setName("Denny");
System.out.println(p1.getName());
```

这段代码中对 People 类的对象引用 p2 的 name 成员变量进行了设置，使其值为字符串"Denny"。但是我们会发现在输出 p1 的成员变量 name 时并不是输出"David"，而是"Denny"。原因是 p1 与 p2 均是对对象的引用，在完成赋值语句" People p2 = p1; "后，p2 与 p1 指向同一存储空间，因此对于 p2 的修改自然会影响到 s1。通过图 1-4 可以清楚说明这段代码运行的情况。

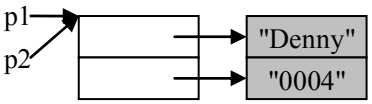
People p1 = new People("David","0004");



People p2 = p1;



p2.setName("Denny");



System.out.println(p1.getName()); 输出：Denny

图 1-4 两个对象引用变量指向同一存储空间

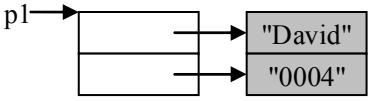
请读者继续考虑以下代码的运行结果：

```
People p1 = new People("David","0004");
People p2 = new People("David","0004");
System.out.println(p1= = p2);
```

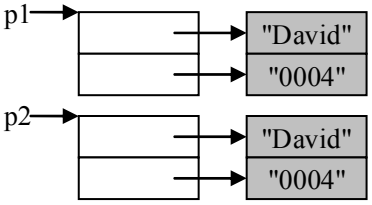
在这里虽然 p1 与 p2 的所有成员变量的内容均相同，但是由于它们指向不同的存储空间，

因此，输出语句输出的结果为 false。图 1-5 说明了 p1 与 p2 的指向。

People p1 = new People("David","0004");



People p1 = new People("David","0004");



System.out.println(p1= = p2);　　输出：false

图 1-5 p1 与 p2 指向不同存储空间

可见如果我们希望完成对象的拷贝，使用一个简单的赋值语句是无法完成的。要达到这一目的可以通过实现 Cloneable 接口并重写 clone 方法来完成。如果我们希望判断两个对象引用是否一致时可以覆盖继承自 Object 类的 equals 方法来实现。

周鹏

第二章　数据结构与算法基础

这一章主要由两部分内容组成：即数据结构和算法的基础知识。在这一章中我们主要介绍数据结构与算法的一些相关基本概念。使读者了解什么是数据结构，数据结构研究的主要内容是什么；同时使读者了解什么是算法，以及如何评价一个算法的性能。

2.1 数据结构

人们在使用计算机解决客观世界中存在的具体问题时，通常过程如下：首先通过对客观世界的认知形成印象和概念从而得到了信息，在此基础上建立概念模型，它必须能够如实地反映客观世界中的事物以及事物间的联系；根据概念模型将实际问题转化为计算机能够理解的形式，然后设计程序；用户通过人机交互界面与系统交流，使系统执行相应操作，最后解决实际的问题。

数据结构主要与在上述过程中从建立概念模型到实现模型转化并为后续程序设计提供基础的内容相关。它是用来反映一个概念模型的内部构成，即一个概念模型由那些成分数据构成，以什么方式构成，呈现什么结构。数据结构主要是研究程序设计问题中计算机的操作对象以及它们之间的关系和操作的学科。

2.1.1 基本概念

在这一小节中首先介绍一些基本概念和术语。

数据（data）是描述客观事物的数值、字符以及能输入机器且能被处理的各种符号集合。数据的含义非常广泛，除了通常的数值数据、字符、字符串是数据以外，声音、图像等一切可以输入计算机并能被处理的都是数据。例如除了表示人的姓名、身高、体重等的字符、数字是数据，人的照片、指纹、三维模型、语音指令等也都是数据。

数据元素（data element）是数据的基本单位，是数据集合的个体，在计算机程序中通常作为一个整体来进行处理。例如一条描述一位学生的完整信息的数据记录就是一个数据元素；空间中一点的三维坐标也可以是一个数据元素。数据元素通常由若干个数据项组成，例如描述学生相关信息的姓名、性别、学号等都是数据项；三维坐标中的每一维坐标值也是数据项。数据项具有原子性，是不可分割的最小单位。

数据对象（data object）是性质相同的数据元素的集合，是数据的子集。例如一个学校的所有学生的集合就是数据对象，空间中所有点的集合也是数据对象。

数据结构（data structure）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。是组织并存储数据以便能够有效使用的一种专门格式，它用来反映一个数据的内部构成，即一个数据由那些成分数据构成，以什么方式构成，呈什么结构。

由于信息可以存在于逻辑思维领域，也可以存在于计算机世界，因此作为信息载体的数据同样存在于两个世界中。表示一组数据元素及其相互关系的数据结构同样也有两种不同的表现形式，一种是数据结构的逻辑层面，即数据的**逻辑结构**；一种是存在于计算机世界的物理层面，即数据的**存储结构**。

数据的逻辑结构按照数据元素之间相互关系的特性来分，可以分为以下四种结构：**集合、**

线性结构、树形结构和**图状结构**。本书中讨论的数据结构主要有线性表、栈、队列、树和图，其中线性表、栈、队列属于线性结构，树和图属于非线性结构。

数据的逻辑结构可以采用两种方法来描述：**二元组、图形**。

数据结构的二元组表示形式为：

数据结构 = {D , S}

其中 D 是数据元素的集合；S 是 D 中数据元素之间的关系集合，并且数据元素之间的关系是使用序偶来表示的。序偶是由两个元素 x 和 y 按一定顺序排列而成的二元组，记作<x , y>，x 是它的第一元素，y 是它的第二元素。

当使用图形来表示数据结构时，是用图形中的点来表示数据元素，用图形中的弧来表示数据元素之间的关系。如果数据元素 x 与 y 之间有关系<x , y>，则在图形中有从表示 x 的点出发到达表示 y 的点的一条弧。

例 2-1 一种数据结构的二元组表示为 set = (K,R)，其中

K = {01, 02, 03, 04, 05}

R = {}

可以看到在数据结构 set 中，只有数据元素的集合非空，而数据元素之间除了同属一个集合之外不存在任何关系（关系集合为空）。这表明该结构只考虑数据元素而不考虑它们之间的关系。我们把具有这种特点的数据结构称为集合结构。

数据结构 set 的图形表示方法见图 2-1。

例 2-2 一种数据结构的二元组表示为 linearity = (K,R)，其中

K = {01, 02, 03, 04, 05}

R = {<02,04>, <03,05>, <05,02>, <01,03>}

可以看到在数据结构 linearity 中，数据元素之间是有序的。在这些数据元素中有一个可以被称为“第一个”（元素 01）的数据元素；还有一个可以被称为“最后一个”（元素 04）的数据元素；除第一个元素以外每个数据元素有且仅有一个直接前驱元素，除最后一个元素以外每个数据元素有且仅有一个直接后续元素。这种数据结构的特点是数据元素之间是 1 对 1 的联系，即线性关系，我们把具有此种特点的数据结构称为**线性结构**。

数据结构 linearity 的图形表示方法见图 2-1。

例 2-3 一种数据结构的二元组表示为 tree = (K,R)，其中

K = {01, 02, 03, 04, 05, 06}

R = {<01,02>, <01,03>, <02,04>, <02,05>, <03,06>}

可以看到在数据结构 tree 中，除了一个数据元素（元素 01）以外每个数据元素有且仅有一个直接前驱元素，但是可以有多个直接后续元素。这种数据结构的特点是数据元素之间是 1 对 N 的联系，我们把具有此种特点的数据结构称为**树结构**。

数据结构 tree 的图形表示方法见图 2-1。

例 2-4 一种数据结构的二元组表示为 graph = (K,R)，其中

K = {01, 02, 03, 04, 05}

R = {<01,02>, <01,05>, <02,01>, <02,03>, <02,04>, <03,02>,
<04,02>, <04,05>, <05,01>, <05,04>}

可以看到在数据结构 graph 中，每个数据元素可以有多个直接前驱元素，也可以有多个直接后续元素。这种数据结构的特点是数据元素之间是 M 对 N 的联系，我们把具有此种特点的数据结构称为**图结构**。

数据结构 graph 的图形表示方法见图 2-1。

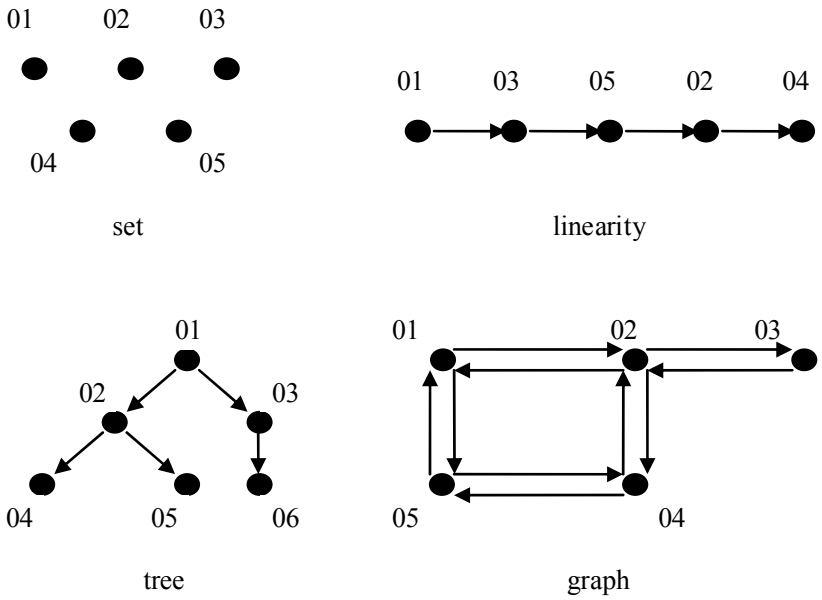


图 2-1 set、linearity、tree、graph 的图形表示

数据的存储结构主要包括数据元素本身的存储以及数据元素之间关系表示。通过数据元素的定义可以看出，我们可以很容易的使用 Java 中的一个类来实现它，数据元素的数据项就是类的成员变量。

数据元素之间的关系在计算机中主要有两种不同的表示方法：**顺序映像**和**非顺序映像**，并由此得到两种不同的存储结构：顺序存储结构和链式存储结构。顺序存储结构的特点是：数据元素的存储对应于一块连续的存储空间，数据元素之间的前驱和后续关系通过数据元素在存储器中的相对位置来反映。链式存储结构的特点是：数据元素的存储对应的是不连续的存储空间，每个存储节点对应一个需要存储的数据元素。元素之间的逻辑关系通过存储节点之间的链接关系反映出来。

由于我们是在 Java 这种计算机高级程序设计语言的基础上来讨论数据结构，因此，我们在讨论数据的存储结构时不会在真正的物理地址的基础上去讨论顺序存储和链式存储，而是在 Java 语言提供的**一维数组**以及**对象的引用**的基础上去讨论和实现数据的存储结构。关于 Java 中的一维数组和对象的引用我们已经在第一章 1.1.4 和 1.4 中分别进行了介绍，在这里不再赘述。

2.1.2 抽象数据类型

抽象数据类型是描述数据结构的一种理论工具。在介绍抽象数据类型之前我们先介绍一下数据类型的基本概念。

数据类型（data type）是一组性质相同的数据元素的集合以及加在这个集合上的一组操作。例如 Java 语言中就有许多不同的数据类型，包括数值型的数据类型、字符串、布尔型等数据类型。以 Java 中的 int 型为例，int 型的数据元素的集合是[-2147483648,2147483647]间的整数，定义在其上的操作有加、减、乘、除四则运算，还有模运算等。

定义数据类型的作用一个是隐藏计算机硬件及其特性和差别，使硬件对于用户而言是透明的，即用户可以不关心数据类型是怎么实现的而可以使用它。定义数据类型的另一个作用

是，用户能够使用数据类型定义的操作，方便的实现问题的求解。例如，用户可以使用 Java 定义在 int 型的加法操作完成两个整数的加法运算，而不用关心两个整数的加法在计算机中到底是如何实现的。这样不但加快了用户解决问题的速度，也使得用户可以在更高的层面上考虑问题。

与机器语言、汇编语言相比，高级语言的出现大大地简便了程序设计。但是要将解答问题的步骤从非形式的自然语言表达到形式化的高级语言表达，仍然是一个复杂的过程，仍然要做很多繁杂琐碎的事情，因而仍然需要抽象。

对于一个明确的问题，要解答这个问题，总是先选用该问题的一个数据模型。接着，弄清该问题所选用的数据模型在已知条件下的初始状态和要求的结果状态，以及隐含着的两个状态之间的关系。然后探索从数据模型的已知初始状态出发到达要求的结果状态所必需的运算步骤。

我们在探索运算步骤时，首先应该考虑顶层的运算步骤，然后再考虑底层的运算步骤。所谓顶层的运算步骤是指定义在数据模型级上的运算步骤，或叫宏观运算。它们组成解答问题步骤的主干部分。其中涉及的数据是数据模型中的一个变量，暂时不关心它的数据结构；涉及的运算以数据模型中的数据变量作为运算对象，或作为运算结果，或二者兼而为之，简称为定义在数据模型上的运算。由于暂时不关心变量的数据结构，这些运算都带有抽象性质，不含运算的细节。所谓底层的运算步骤是指顶层抽象的运算的具体实现。它们依赖于数据模型的结构，依赖于数据模型结构的具体表示。因此，底层的运算步骤包括两部分：一是数据模型的具体表示；二是定义在该数据模型上的运算的具体实现。我们可以把它们理解为微观运算。于是，底层运算是顶层运算的细化，底层运算为顶层运算服务。为了将顶层算法与底层算法隔开，使二者在设计时不会互相牵制、互相影响，必须对二者的接口进行一次抽象。让底层只通过这个接口为顶层服务，顶层也只通过这个接口调用底层的运算。这个接口就是抽象数据类型。

抽象数据类型（abstract data type，简称 ADT）由一种数据模型和在该数据模型上的一组操作组成。

抽象数据类型包括定义和实现两个方面，其中定义是独立于实现的。抽象数据类型的定义仅取决于它的逻辑特性，而与其在计算机内部的实现无关，即无论它的内部结构如何变化，只要它的逻辑特性不变，都不会影响到它的使用。其内部的变化（抽象数据类型实现的变化）只是可能会对外部在使用它解决问题时的效率上产生影响，因此我们的一个重要任务就是如何简单、高效地实现抽象数据类型。很明显，对于不同的运算组，为使组中所有运算的效率都尽可能地高，其相应的数据模型具体表示的选择将是不同的。在这个意义下，数据模型的具体表示又依赖于数据模型上定义的那些运算。特别是，当不同运算的效率互相制约时，还必须事先将所有的运算的相应使用频度排序，让所选择的数据模型的具体表示优先保证使用频度较高的运算有较高的效率。

我们应该看到，抽象数据类型的概念并不是全新的概念。抽象数据类型和数据类型在实质上是一个概念，只不过是对数据类型的进一步抽象，不仅限于各种不同的计算机处理器中已经实现的数据类型，还包括为解决更为复杂的问题而由用户自定义的复杂数据类型。例如高级语言都有的“整数”类型就是一种抽象数据类型，只不过高级语言中的整型引进实现了，并且实现的细节可能不同而已。我们没有意识到抽象数据类型的概念已经孕育在基本数据类型的概念之中，是因为我们已经习惯于在程序设计中使用基本数据类型和相关的运算，没有进一步深究而已。

抽象数据类型一方面使得使用它的人可以只关心它的逻辑特征，不需要了解它的实现方式。另一方面可以使我们更容易描述现实世界，使得我们可以在更高的层面上来考虑问题。例如可以使用树来描述行政区划，使用图来描述通信网络。

根据抽象数据类型的概念，对抽象数据类型进行定义就是约定抽象数据类型的名字，同时，约定在该类型上定义的一组运算的各个运算的名字，明确各个运算分别要 有多少个参数，这些参数的含义和顺序，以及运算的功能。一旦定义清楚，人们在使用时就可以像引用基本数据类型那样，十分简便地引用抽象数据类型；同时，抽象数据类型的实现就有了设计的依据和目标。抽象数据类型的使用和实现都与抽象数据类型的定义打交道，这样使用与实现没有直接的联系。因此，只要严格按照定义，抽象数据类型的使用和实现就可以互相独立，互不影响，实现对它们的隔离，达到抽象的目的。

为此抽象数据类型可以使用一个三元组来表示：

ADT = (D, S, P)

其中 D 是数据对象，S 是 D 上的关系集，P 是加在 D 上的一组操作。

在定义抽象数据类型时，我们使用以下格式：

ADT 抽象数据类型名{

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

}

2.1.3 小结

通过以上两小节的内容我们可以看到数据结构就是研究三个方面的主要问题的：数据的逻辑结构、数据的存储结构以及定义在数据结构上的一组操作。即研究按照某种逻辑关系组织起来的一批数据，并按一定的映像方式把它们存放在计算机的存储器中，最后分析在这些数据上定义的一组操作。为此我们要考虑怎样合理的组织数据，建立合适的结构，提高实现的效率。

在数据结构的实现中我们可以很好的将数据结构中的一些基本概念和 Java 语言中的一些概念对应起来。数据元素可以对应到类，其数据项就是类的成员变量，某个具体的数据元素就是某个类的一个实例；数据的顺序存储结构与链式存储结构可以通过一维数组以及对象的引用来实现；抽象数据类型也可以对应到类，抽象数据类型的数据对象与数据之间的关系可以通过类的成员变量来存储和表示，抽象数据类型的操作则使用类的方法来实现。

2.2 算法及性能分析

算法设计是最具创造性的工作之一，人们解决任何问题的思想、方法和步骤实际上都可以认为是算法。人们解决问题的方法有好有坏，因此算法在性能上也就有高低之分。在这一节中我们首先给出算法的定义，然后介绍分析算法性能的理论方法。

2.2.1 算法

算法（algorithm）是指令的集合，是为解决特定问题而规定的一系列操作。它是明确定义的可计算过程，以一个数据集合作为输入，并产生一个数据集合作为输出。一个算法通常来说具有以下五个特性：

- 输入：一个算法应以待解决的问题的信息作为输入。

- 输出：输入对应指令集处理后得到的信息。
- 可行性：算法是可行的，即算法中的每一条指令都是可以实现的，均能在有限的时间内完成。
- 有穷性：算法执行的指令个数是有限的，每个指令又是在有限时间内完成的，因此整个算法也是在有限时间内可以结束的。
- 确定性：算法对于特定的合法输入，其对应的输出是唯一的。即当算法从一个特定输入开始，多次执行同一指令集结果总是相同的。

对于随机算法，算法的第五个特性应当被放宽。在本书中所讨论的算法均是确定性算法。

简单的说，算法就是计算机解题的过程。在这个过程中，无论是形成解题思路还是编写程序，都是在实施某种算法。前者是算法的逻辑形式，后者是算法的代码形式。

2.2.2 时间复杂性

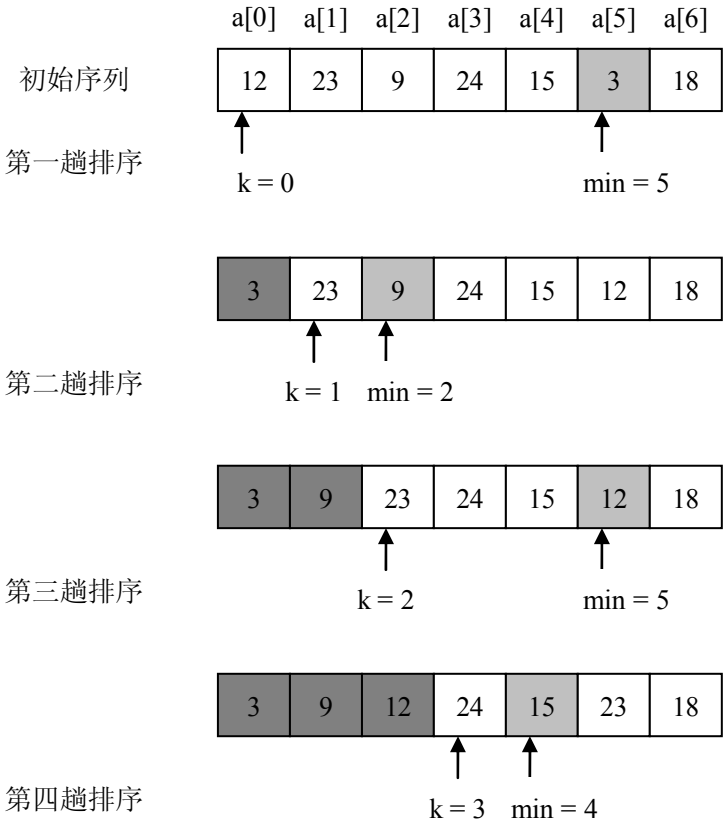
在计算机资源中，最重要的就是时间与空间。评价一个算法性能的好坏，实际上就是评价算法的资源占用问题。在这一小节中我们讨论算法运行时间的确定问题，这个问题被称之为算法的时间复杂性。关于算法的空间性能评价我们将在 2.2.3 中介绍。

本节以一个例子开始，通过它来说明如何去分析算法的运行时间。

例 2-5　简单选择排序：

令 A[0,n-1]为有 n 个数据元素的数组，我们的目标是将数组 A 排序为一个非降序的有序数组。使用简单选择排序来解决这个问题的算法是：首先在 n 个元素中找到最小元素，将其放在 A[0]中，然后在剩下的 n-1 个元素中找到最小的放到 A[1]中，这个过程不断进行下去，直到在最后 2 个元素中找到小的并将其放到 A[n-2]中。

图 2-2 说明了对具有 7 个整数的数组使用简单选择排序的过程。



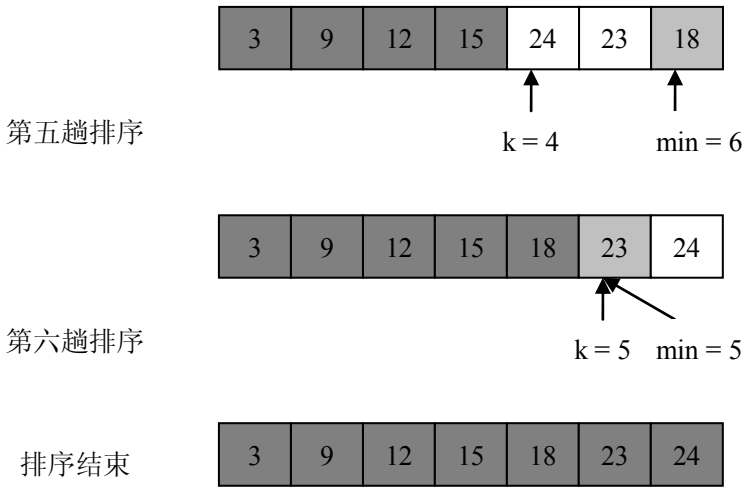


图 2-2 对 7 个整数简单选择排序过程

算法 2-1 selectSort
输入：整型数组 a[0,n-1]
输出：按非降序排列的数组 a[0,n-1]
代码：

```
public void selectSort (int[] a) {
    int n = a.length;
    for (int k=0; k<n-1; k++) {
        int min = k;
        for (int i=k+1; i<n; i++)
            if (a[i]<a[min]) min = i;
        if (k!=min){
            int temp = a[k];
            a[k] = a[min];
            a[min] = temp;
        }//end of if
    }//end of for
}
```

通过对算法的分析可以看出，这个算法执行的比较次数为

$$\sum_{i=1}^{n-1}(n-i) = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

同时也可以看出数据元素交换的次数在 0 到 n-1 之间，而每次交换需要使用 3 条赋值语句，因此数据元素的赋值在 0 到 3(n-1)之间。

首先我们说一个算法对于某个输入要用 x 秒运行是没有意义的。这是因为影响实际运行时间的因素不仅有算法本身，还有其他诸多因素。例如算法是在什么机器上运行的，不同的机器其运算速度是不一样的；除了硬件的影响，操作系统以及使用的高级语言、编译系统都会对算法的实际运行时间造成影响。因此在对算法的运行时间做出分析时我们应该避免这些因素的影响，为此我们可以假设一些基本操作都是可以在一个常数时间内完成的。例如逻辑

运算、赋值运算等都是基本操作。这样算法执行基本操作的次数可以反映算法的运行时间，在后面提到算法的运行时间时都是指运行基本操作的次数。

其次，即使能够排除软硬件的影响，对同一个算法而言，如果问题的规模不同，那么实际的运算时间也会有很大差异。例如在算法 2-1 中，如果算法分别对 100 个整数排序以及对10^9个整数排序，其实际运行时间的差异是非常大的。假设算法中进行一比较需要 10^{-8}秒，那么对 100 个数进行排序大概需要 $100*99/2*10^{-8}=0.0000495$ 秒，而对 10^9个数排序则需要$10^9*(10^9-1)/2*10^{-8}=158.5$ 年。为此在分析算法的运行时间时我们必须将问题的规模（通常用n来表示）也考虑进去。显然算法执行基本操作的次数是关于规模n的非负函数，我们把它记为T(n)。

下面给出了一些常用的作为问题规模的例子：

- 在排序和查找问题中，用数组或表中元素的个数作为问题的规模；
- 在图的相关问题中，用图中的点或（和）边的数目；
- 在计算几何问题中，用顶点、边、线段或多边形的数目；
- 在矩阵运算中，用矩阵的维数；
- 在数论问题中，通常将表示输入数的比特数来表示输入大小。

最后，在分析一个算法在输入一个规模为 n 的实例时，我们是否需要将 T(n)的精确值求出来呢？

事实上我们在评估一个算法时并不需要精确计算 T(n)。因为这是没有必要并且有时也是不可能的。算法的性能好坏是通过与其他算法的比较而得出的，由于算法对小规模输入实例所需的处理时间很少，因此小规模输入实例对性能的比较没有多大意义。我们关注的是在问题规模很大时算法的效率差异，而当问题的规模 n 不断变大时，在函数 T(n)中有一些部分就会变得不重要。例如在算法 2-1 中当 n 不断变大时，算法执行的所有赋值语句对整个运行时间的影响就越来越小。为此我们可以将这些不重要的部分忽略，转而讨论运行时间的增长率或增长的阶。

一旦去掉表示算法运行时间中的低阶项和首项常数，就称我们是在度量算法的**渐进时间复杂度（asymptotic complexity）**，简称**时间复杂度**。

为了进一步说明算法的时间复杂度，我们定义Ο、Ω、Θ符号。

■ Ο符号

对算法 2-1 的 T(n)进行分析，我们可以得到：

$$T(n) \le \frac{n(n-1)}{2} + 3(n-1) \le cn^2 \quad （c 为某个正常数）$$

这时我们说算法 2-1 的时间复杂度为Ο(n^2)，这个符号可以解释为：只要当排序元素的个数大于某个阈值N时，那么对于某个常量$c > 0$，运行时间最多为cn^2。也就是说Ο符号提供了一个运行时间的上界。

定义 2-1 令 T(n)和 f(n)是非负函数，如果存在一个非负整数 N 以及一个常数 c>0，使得：

$$\forall n \ge N, T(n) \le cf(n)$$

则 T(n) = Ο(f(n))。

即如果$\lim_{n\to\infty} T(n)/f(n)$存在，那么

$$\lim_{n\to\infty}\frac{T(n)}{f(n)}\neq\infty\Rightarrow\ T(n)=\ O(f(n))$$

例如，算法 2-1 的时间复杂度 $T(n)\le\frac{n(n-1)}{2}+3(n-1)$，由于当 $n\ge 2$ 时，$T(n)\le 2n^2$，则有 $T(n)=O(n^2)$。或令 $f(n)=n^2$，因为 $\lim_{n\to\infty}\frac{T(n)}{f(n)}\le\lim_{n\to\infty}\frac{n(n-1)/2+3(n-1)}{n^2}=\frac{1}{2}\neq\infty$，则有 $T(n)=O(n^2)$。

■ Ω 符号

O 符号给出了算法时间复杂度的上界，而 Ω 符号在运行时间的常数因子范围内给出了时间复杂度的下界。

Ω 符号可以解释为：如果输入大于等于某个阈值 N，算法的运行时间下限是 f(n)的 c 倍，其中 c 是一个正常数，则称算法的时间复杂度是 Ω(f(n))的。Ω 的形式定义与 O 符号对称。

定义 2-2 令 T(n)和 f(n)是非负函数，如果存在一个非负整数 N 以及一个常数 c>0，使得：

$$\forall n\ge N, T(n)\ge cf(n)$$

则 T(n)= Ω(f(n))。

即如果 $\lim_{n\to\infty}T(n)/f(n)$ 存在，那么

$$\lim_{n\to\infty}\frac{T(n)}{f(n)}\neq 0\Rightarrow\ T(n)=\ \Omega(f(n))$$

例如，算法 2-1 的时间复杂度 $T(n)\ge\frac{n(n-1)}{2}$，由于当 $n\ge 2$ 时，$T(n)\ge\frac{1}{4}n^2$，则有 $T(n)=\Omega(n^2)$。或令 $f(n)=n^2$，因为 $\lim_{n\to\infty}\frac{T(n)}{f(n)}\ge\lim_{n\to\infty}\frac{n(n-1)/2}{n^2}=\frac{1}{2}\neq 0$，因此则有 $T(n)=\Omega(n^2)$。

■ θ 符号

O 符号给出了算法时间复杂度的上界，Ω 符号给出了时间复杂度的下界，而 θ 给出了算法时间复杂度的精确阶。

Θ 符号可以解释为：如果输入大于等于某个阈值N，算法的运行时间在下限c_1f(n)和上限c_2f(n)之间（$0<c_1\le c_2$），则称算法的时间复杂度是 Θ(f(n))阶的。该符号的形式定义如下。

定义 2-3 令T(n)和f(n)是非负函数，如果存在一个非负整数N以及常数$c_1>0$ 和$c_2>0$，使

得：

$$\forall n \geq N, c_1 f(n) \leq T(n) \leq c_2 f(n)$$

则 T(n) = Θ(f(n))。

即如果 $\lim_{n\to\infty} T(n)/f(n)$ 存在，那么

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = c\ (c > 0) \Rightarrow \ T(n) = \Theta(f(n))$$

例如，算法 2-1 的时间复杂度 $\frac{n(n-1)}{2} \leq T(n) \leq \frac{n(n-1)}{2} + 3(n-1)$，由于当 $n \geq 2$ 时，

$\frac{1}{4}n^2 \leq T(n) \leq 2n^2$，则有 $T(n) = \Theta(n^2)$。或令 $f(n) = n^2$，因为 $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \frac{1}{2}$，则有

$T(n) = \Theta(n^2)$。

定义 2-3 的一个重要推论是

$T(n) = \Theta(n^2)$，当且仅当 $T(n) = O(n^2)$ 并且 $T(n) = \Omega(n^2)$

通过以上的分析我们可以看出：我们评价算法的运行时间是通过分析在一定规模下算法执行基本操作的次数来反映的，并且由于我们只对大规模问题的运行时间感兴趣，所以是使用算法的渐进时间复杂度 T(n)来度量算法的时间性能的。O、Ω、Θ 符号分别定义了时间复杂度的上界、下界以及精确阶。

2.2.3 空间复杂性

在上一小节中我们讨论了算法的时间复杂性，下面我们来讨论算法的空间复杂性。

算法的空间复杂性同样是由算法运行时使用的空间来评价的。我们把算法使用的空间定义为：为了求解问题的实例而执行的操作所需要的存储空间的数目，但是它不包括用来存储输入实例的空间。

同样前面对于评价算法时间复杂性的讨论都可以用于对算法的空间复杂性的讨论。并且在这里有这样一个观察结论：算法的空间复杂性是以时间复杂性为上界的。这是因为在算法中每访问一次存储空间都是需要使用一定时间的，即使每次访问的都是不同的存储空间，空间的大小也不会超过基本操作次数常数倍，因此算法的空间复杂性是以时间复杂性为上界的。

如果使用 S(n)与 T(n)分别表示算法的空间复杂度和时间复杂度，则有 $S(n) = O(T(n))$。

例如在算法 2-1 中，为了算法的执行，我们使用了常数个中间变量，每个变量的存储空间都是常数大小，所以在算法 2-1 中：$S(n) = O(1) = \Omega(1) = \Theta(1)$

2.2.4 算法时间复杂度分析

我们不但要了解什么是算法时间复杂度，还要学会分析算法的时间复杂度。最简单的方法就是将算法执行的所有基本操作都计算出来，然后得出算法的时间复杂度。但是很多时候这种方法是不可取的，因为它太麻烦而且可能计算不出所有基本操作的执行次数。

一般来说，不存在固定的方法，使用它就可以得到一个算法的时间复杂度。但是在分析算法的时间复杂度时有一些常用技术是可以使用的。

■ **计算循环次数**

运行时间往往都是集中在循环中，循环之外往往是一些简单的具有常数基本操作的运算，而这些常数次的基本操作在渐进时间复杂度中是不起作用的。这就使得运行时间往往和循环的次数具有相同的阶。

下面给出几个使用这种方法分析算法时间复杂度的例子。

算法 2-2 function1
输入：正整数 n
输出：循环执行的总次数
代码：

```
public int function1 (int n) {
    int i = 1, count = 0;
    while (i <= n)    {i = i * 2; count++; }
    return count;
 }
```

例 2-6　分析上面的算法function1 的时间复杂度。在这里只有一层循环，假设循环k次，循环每进行一次会执行常数个基本操作，因此算法的时间复杂度T(n) = Θ(k)。因为循环只执行了k次，i在循环执行过程中的变化趋势为 1,2,4,8…2^k，在执行完第k次循环后i=2^k，所以有

$$2^{k-1} \le n < 2^k \Rightarrow k = \lfloor \log n \rfloor + 1$$

由此可知 $T(n) = \Theta(k) = \Theta(\lfloor \text{long } n \rfloor + 1) = \Theta(\log n)$ 。

算法 2-3 function2
输入：正整数 n
输出：循环执行的总次数
代码：

```
public int function2 (int n)
{
    int count=0, s=0;
    while (s<n) {count++; s = s + count; }
    return count;
 }
```

例 2-7　分析上面的算法 function2 的时间复杂度。这个算法与算法 function1 类似，算法的时间复杂度与循环次数具有相同的阶。假设 while 循环执行了 k 次，而 count 在 while 循环执行过程中的变化趋势为 0,1,2…k，在执行完第 k 次循环后 i=k。而 s 在第 i(0<i<k+1)次循环后的值为

$s = 0 + 1 + 2 + ... + i = \frac{i(i+1)}{2}$，因为循环只执行了 k 次，所以有

$$\frac{(k-1)k}{2} < n \le \frac{k(k+1)}{2} \Rightarrow k = \Theta(n^{\frac{1}{2}})$$

因此，$T(n) = \Theta(k) = \Theta(n^{\frac{1}{2}})$。

算法 2-4 function3
输入：正整数 n
输出：循环执行的总次数
代码：

```
public int function3 (int n) {
    int i = 1, count = 0;
    while (i <= n)   {
        for (int j = 0; j < i; j++)
            count++;
        i = i * 2;
    }
    return count;
}
```

例 2-8　分析上面的算法function3 的时间复杂度。假设while循环执行了k次，for循环每次执行i次，而i在while循环执行过程中的变化趋势为 1,2,4,8…2^k，在执行完第k次循环后$i=2^k$。所以，循环总的执行次数为

$1 + 2 + ... + 2^{k-1} = \sum_{i=0}^{k-1} 2^i = 2^k - 1$，并且由例 2-6 知 $k = \lfloor \log n \rfloor + 1$

因此，$T(n) = \Theta(2^k - 1) = \Theta(2^{\lfloor \log n \rfloor + 1}) = \Theta(n)$。

■ 分析最高频度的基本操作

在某些算法中，使用统计循环次数的方法来完成算法时间复杂度的估算是非常复杂的，有时甚至是无法完成的。

请读者先阅读下面的例子。

下面的算法是将两个已经有序的数组 a[0,m-1]，b[0,n-1]合并为一个更大的有序数组。合并两个数组的算法是使用两个变量 pa，pb 初始时分别指向数组 a、b 的第一个元素，每次比较 a[pa]与 b[pb]，将小的放入新的数组 c。更新指向小元素的变量，使之加 1，指向后续元素。这个过程一直进行下去，直到将某个数组中的所有元素放入新的数组为止。此时再将另

一个数组中剩余的数据元素依次放入新数组。这一过程在算法 2-5 中给出。

算法 2-5 merge
输入：整型非递减数组 a[0,m-1]，b[0,n-1]
输出：将数组 a 与 b 合并为一个新的非递减数组 c
代码：

```
public int[] merge (int[] a, int[] b) {
    int pa = pb = pc = 0;
    int m = a.length;
    int n = b.length;
    int[] c = new int[m+n];
    while (pa<m && pb<n) {
        if (a[pa]<b[pb])
            c[pc++] = a[pa++];
        else
            [pc++] = b[pb++];
    }
    if (pa<m)
        while (pa<m) c[pc++] = a[pa++];
    else
        while (pb<n) c[pc++] = b[pb++];
    return c;
}
```

例 2-9　分析算法 2-5 的时间复杂度。如果使用分析循环次数的方法来求算法的时间复杂度，在这个例子中会较为复杂，因为这里有三个循环，而每个循环执行的次数会由于输入实例的不同而不同。

在这里我们可以使用分析最高频度的基本操作的方法来得到算法 merge 的时间复杂度。我们可以观察到在算法中主要操作有元素的赋值和比较两种操作，而赋值是使用频度最高的基本造作，因为每个元素都必须要放入新的数组，而并不是每个元素都需要比较才能放入新的数组。在算法中的每个元素至少赋值一次，至多也赋值一次，因此共有 m+n 次赋值，因此算法 merge 的时间复杂度 $T(n) = \Theta(m+n)$。

在这一小节中，我们主要介绍了两种确定算法时间复杂度的基本方法，使用递推公式求解的方法在第五章中介绍。

2.2.5 最佳、最坏与平均情况分析

在上面的例子中，所有算法对于任何输入实例，它的时间复杂度都是相同的。但是有些算法的执行时间不但是规模的函数，也是输入实例数据元素初始顺序的实例。对于不同的输入实例，时间复杂度会有很大的差异，为此我们要对不同的情况进行不同的分析。

下面我们介绍一个非常简单的例子用来说明这种情况。

例如我们需要查找存在于一个数组中的某个元素，最简单的方法就是从左到右依次扫描数组中的每个数据元素，如果发现当前元素和需要查找的数据元素相同，则返回元素在数组中的下标。

算法 2-5 linearSearch

输入：整型数组 a[0,n-1]，整数 k = a[i]。0≤ i < n

输出：i

代码：

```
public int linearSearch (int[] a, int k) {
    for (int i=0; i<n; i++)
        if (a[i] == k)    return i;
    return -1;
 }
```

这段代码非常简单，但是我们却不能武断地说这个算法的时间复杂度就是多少，因为这个算法会因为输入实例的不同，其执行时间会有很大的差异。对此我们作以下分析。

⑴ 如果输入的实例中要寻找的 a[0]就是 k，那么算法执行一次比较操作就会结束返回，此时执行的基本操作的个数为 Θ(1)。这是最好的情况。

⑵ 如果输入的实例中要寻找的 a[n-1]是 k，那么算法需要执行 n 次比较操作才能找到 k，然后结束返回，此时执行的基本操作的个数为 Θ(n)。这是最坏情况。

⑶ 第三种是平均情况，这里的平均情况是指执行所有大小为 n 的输入时算法的平均执行时间。

这三种情况分别对应了时间复杂性的最佳、最坏以及平均情况分析。

在最佳与最坏的情况分析中，分别对应的是我们在所有大小为 n 的输入中选择代价最小和最大的。对于最佳和最坏情况的分析较简单，例如上面我们对算法 linearSearch 的第一和第二种情况的分析。

在平均情况的分析中，我们首先必须知道所有大小为 n 的输入的分布，即要知道每一种情况出现的概率。即使这样，在许多情况下对于算法的平均情况分析也是复杂的。下面我们分析一下算法 linearSearch 执行的平均情况。

例 2-10 算法 linearSearch 的平均情况分析。为了简化讨论，我们假设 a[0,n-1]中元素互不相同。还假定 k 出现在数组中，并且最重要的是，我们假设数组中的任何一个元素出现在数组中任何一个位置上是等概率的。即

$$\forall x \in a, P[x = a[j]] = \frac{1}{n} \quad (0 \le j < n)$$

假设C_j是当k出现在数组下标为j的地方时需要比较的次数，那么$C_j = j+1, 0 \le j < n$。此时，为了找到k的位置，算法执行比较次数的平均值是：

$$T(n) = \sum_{j=0}^{n-1}\left(C_j \cdot P[k = a[j]]\right) = \frac{1}{n}\sum_{j=0}^{n-1}(j+1) = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2} = \Theta(n)$$

这说明在平均情况下，为找到 k 的位置，我们大概要比较数组中一半的数据元素。因此算法 linearSearch 的时间复杂度在平均情况下是 Θ(n)。

由此可见，除了前面我们提到的影响因素，输入实例本身也会对算法时间复杂度的分析造成影响。

2.2.6 均摊分析

均摊分析也是一种非常重要的时间复杂度分析方法，使用这种分析方法经常会给我们对算法时间复杂度的分析带来意想不到的效果，能够让我们为算法找到更小的时间复杂度上界，而这个上界通常会比我们意想中的上界小得多。

如果在一个算法中反复出现具有以下这样特性的计算时，我们就需要考虑使用均摊分析了。这种特性是：计算的运行时间始终变动，并且这一计算在大多数时候运行得很快，而在少数时候却要花费大量时间。

在均摊分析中，我们可以算出算法在整个执行过程中，或多次执行过程中所用时间的平均值，称为该算法的均摊运行时间。均摊分析保证了算法的平均代价，这与平均情况分析是不同的，在平均分析中必须知道每种输入实例的分布，计算所有不同输入实例才能得到平均值，而在均摊分析中不需要这样。

下面我们举两个例子来说明均摊分析的要点以及进行均摊分析的方法。

例 2-11 我们知道数组的大小一旦确定，那么在以后的使用过程中就不能再改变。当要给未知个数的数据元素分配存储空间时，可以采用以下策略：先分配一个初始大小为 m 的数组 A0，当 A0 空间用完之后，在第 m+1 个数据元素进入之前，分配一个大小为 2m 的新的数组 A1，并将 A0 中的元素全部移到 A1 中。如果 A1 空间用完，则再次新建大小为原来空间 2 倍的数组，移动元素到新数组。如此往复，直到所有元素都存入数组为止。

在这里假设 m=1，如果 m>1，则执行赋值的次数要多于从 1 开始的情况，因为在数组大小被扩展到 m 之前是没有元素的移动过程的。因此 m=1 时的运行时间是最多的，它是当 m 取其他值时运行时间的上界。

在算法中使用频度最高的是赋值操作，因此我们对元素赋值的次数进行计数，结果可以反映算法的时间复杂度。

假设加入的数据元素的个数为 n。则每个元素最多被移动 n 次，所以 n 个元素加入数组总的时间 $T(n)=O(n^2)$。这是加入 n 个元素运行时间的上界，但是这个上界不够紧密，我们可以通过均摊分析得到更紧密的上界。分析如下。

当 m=1 时，算法的执行情况可用图 2-3 表示。

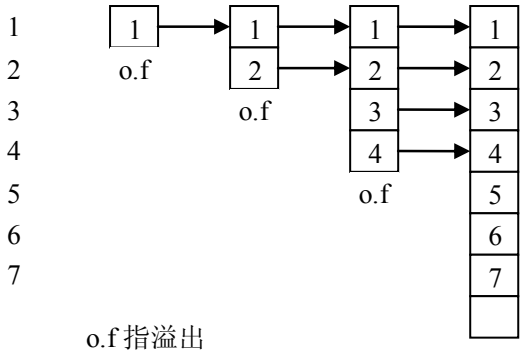


图 2-3 数组倍增过程

为计算加入n个元素需要元素赋值的次数，先定义变量C_i，其值为加入第i个元素时元素赋值的次数。通过图 2-3 可以得出

$$C_i = \begin{cases} i & i-1 \text{ 为 2 的整数次幂} \\ 1 & \text{其他} \end{cases}$$

表 2-1 反映了图 2-3 所示过程中数组大小及C_i变化的情况。

得到：$\sum_{j=1}^{n} C_j = n + \sum_{i=0}^{\lfloor \log(n-1) \rfloor} 2^i \le 3n$，因此 n 个元素加入数组总的时间 $T(n) = \Theta(n) << O(n^2)$。这个时间均摊到每个元素，则存储和移动每个元素代价是 $\Theta(1)$ 均摊时间。

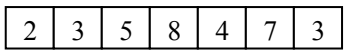
i	1	2	3	4	5	6	7	8	9	10
size	1	2	4	4	8	8	8	8	16	16
C_i	1	2	3	1	5	1	1	1	9	1
C_i	1	1 1	1 2	1	1 4	1	1	1	1 8	1

表 2-1 C_i 随 i 变化情况

有时在对一个算法进行均摊分析时，不能像上面那样能够求出每次计算的时间，因此无法使用求和的方法来得出 n 次计算的时间总和，然后再均摊到每次计算上。这时可以使用资源预留的方法进行分析。

例 2-12 考虑下面的算法。有一个 n 个正整数的数组 a[0, n-1]作为输入，同时生成一个大小与 a 相同的数组 array，然后依次处理 a 中每个元素：如果当前的 a[i]是奇数则直接添加到 array 中最后一个元素后面；如果是偶数，则从 array 中最后一个元素开始，向前依次删除所有的奇数。这个过程可以通过图 2-4 说明。

输入数组 a



数组 array 变化情况：

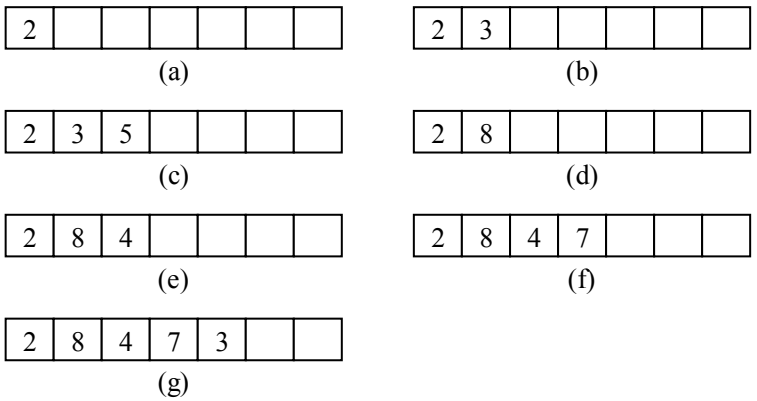


图 2-4 数组 a 元素处理过程

代码：

```
public void function4 (int[] a ) {
    int p = 0, n = a.length;
    int[] array = new int[n];

    for (int i=0; i<n; i++){
        if (a[i]%2 == 0)      //如果是偶数
            while (p>0 && array[p-1]%2!=0)  array[p--] = 0; //删除前面的奇数
        array[p++] = a[i];
    }
    return ;
}
```

现在来分析一下算法的时间复杂度。在某些情况下，例如n/2 个奇数后面跟着一个偶数，那么while循环要执行 $O(n)$次，这里for循环要执行n次，因此总的运行时间是 $O(n^2)$。同样如果我们使用均摊分析，可以得出复杂性为 $\Theta(n)$。

这里共有这样一些基本操作：添加、删除元素。但是在每一次for循环执行过程中不知道到底删除了多少个数据元素，因此无法如同例 2-11 那样计算出C_i，也就无法直接求出n次计算执行基本操作的总次数。

此时我们可以采用资源预留的分析方法，在进行每一次计算时，假设我们都为该次计算预留一定的时间。假设第 i 次计算执行 $C_i^{'}$ 次基本操作，只要保证不等式 $\sum_{i=1}^{n} C_i \le \sum_{i=1}^{n} C_i^{'}$ 成立，则 $\sum_{i=1}^{n} C_i^{'}$ 就是算法的时间上界。

在本例中可以设 $C_i^{'} = 2$，其中一次操作用于元素的添加操作，一次用于元素的删除操作。由于偶数只执行添加操作，而每个奇数最多进行一次添加和一次删除操作，因此 $\sum_{i=1}^{n} C_i \le \sum_{i=1}^{n} C_i^{'} = 2n$， $T(n) = \Theta(n)$。把这个时间均摊到每个元素上，则每个元素的操作时间为 $\Theta(1)$，即 while 语句的均摊时间是 $\Theta(1)$。

第三章　线性表

线性结构是最简单，也是最常用的数据结构之一。线性结构的特点是：在数据元素的有限集中，除第一个元素无直接前驱，最后一个元素无直接后续以外，每个数据元素有且仅有一个直接前驱元素和一个直接后续元素。在这一章中主要介绍线性表的基本概念、定义线性表的抽象数据类型；在给出线性表的顺序存储结构和链式存储结构的基础上，分别给出线性表抽象数据类型的实现。

3.1 线性表及抽象数据类型

3.1.1 线性表定义

线性表(linear list)是n个类型相同数据元素的有限序列，通常记作(a0, a1, …ai-1, ai, ai+1 …, an-1)。

在线性表的定义中，我们看到从a0到an-1的n个数据元素是具有相同属性的元素。比如说可以都是数字，例如(23, 14, 66, 5, 99)；也可以是字符，例如(A, B, C, … Z)；当然也可以是具有更复杂结构的数据元素，例如每个数据元素可以是我们在前面定义过的学生这种类的一个实例。

在线性表的相邻数据元素之间存在着序偶关系，即ai-1是ai的直接前驱，则ai是ai-1的直接后续，同时ai又是ai+1的直接前驱，ai+1是ai的直接后续。唯一没有直接前驱的元素a0一端称为表头，唯一没有后续的元素an-1一端称为表尾。

线性表中数据元素的个数n定义为线性表的长度，当n=0 时线性表为**空表**。在非空的线性表中每个数据元素在线性表中都有唯一确定的**序号**，例如a0的序号是 0，ai的序号是i。在一个具有n > 0 个数据元素的线性表中，数据元素序号的范围是[0, n-1]。

在这里特别需要注意的是线性表和数组的区别。从概念上来看，线性表是一种抽象数据类型；数组是一种具体的数据结构。线性表与数组的逻辑结构是不一样的，线性表是元素之间具有 1 对 1 的线性关系的数据元素的集合，而数组是一组数据元素到数组下标的一一映射。并且从物理性质来看，数组中相邻的元素是连续地存储在内存中的；线性表只是一个抽象的数学结构，并不具有具体的物理形式，线性表需要通过其它有具体物理形式的数据结构来实现。在线性表的具体实现中，表中相邻的元素不一定存储在连续的内存空间中，除非表是用数组来实现的。对于数组，可以利用其下标在一个操作内随机存取任意位置上的元素；对于线性表，只能根据当前元素找到其前驱或后继，因此要存取序号为i的元素，一般不能在一个操作内实现，除非表是用数组实现的。

线性表是一种非常灵活的数据结构，线性表可以完成对表中数据元素的访问、添加、删除等操作，表的长度也可以随着数据元素的添加和删除而变化。

3.1.2 线性表的抽象数据类型

下面我们给出线性表的抽象数据类型定义。

ADT List {

数据对象：D = {a_i | a_i∈D0，i=0, 1, 2…n-1，D_0为某一数据对象}

数据关系：R = {<a_i, a_{i+1}> | a_i, a_{i+1}∈D，i=0, 1, 2 … n-2}

基本操作：

序号	方法	功能描述
⑴	getSzie ()	输入参数：无 返回参数：非负整数 功能：返回线性表的大小，即数据元素的个数。
⑵	isEmpty ()	输入参数：无 返回参数：boolean 功能：如果线性表为空返回 true，否则返回 false。
⑶	contains (e)	输入参数：Object 对象 e 返回参数：boolean 功能：判断线性表是否包含数据元素 e，包含返回 true，否则返回 false。
⑷	indexOf (e)	输入参数：Object 对象 e 返回参数：整数 功能：返回数据元素 e 在线性表中的序号。如果 e 不存在则返回-1。
⑸	insert (i, e)	输入参数：非负整数 i（序号），Object 对象 e 返回参数：无 功能：将数据元素 e 插入到线性表中 i 号位置。若 i 越界，报错。
⑹	insertBefore (p, e)	输入参数：Object 对象 p，Object 对象 e 返回参数：boolean 功能：将数据元素 e 插入到元素 p 之前。成功返回 true，否则返回 false。
⑺	insertAfter (p, e)	输入参数：Object 对象 p，Object 对象 e 返回参数：boolean 功能：将数据元素 e 插入到元素 p 之后。成功返回 true，否则返回 false。
⑻	remove (i)	输入参数：非负整数 i（序号） 返回参数：Object 对象 功能：删除线性表中序号为 i 的元素，并返回之。若 i 越界，报错。
⑼	remove (e)	输入参数：Object 对象 e 返回参数：boolean 功能：删除线性表中第一个与 e 相同的元素。成功返回 true，否则返回 false。
⑽	replace (i, e)	输入参数：非负整数 i（序号），Object 对象 e 返回参数：Object 对象 功能：替换线性表中序号为 i 的数据元素为 e，返回原数据元素。若 i 越界，报错。
⑾	get (i)	输入参数：非负整数 i（序号） 返回参数：Object 对象 功能：返回线性表中序号为 i 的数据元素。若 i 越界，报错。

} ADT List

在上述抽象数据类型的定义中，我们定义了 11 种操作，然而对于线性表的操作并不仅限于上述的操作，根据实际情况的需要还可以定义更多更复杂的操作。例如，将两个线性表合并为一个更大的线性表；把一个线性表分成两个线性表；对现有线性表进行复制等。

3.1.3 List 接口

通过 2.1.3 中的内容我们知道：抽象数据类型可以对应到 Java 中的类，抽象数据类型的数据对象与数据之间的关系可以通过类的成员变量来存储和表示，抽象数据类型的操作则使用类的方法来实现。

下面先不考虑如何完成数据对象以及数据之间关系的存储和表示，我们考虑如何将抽象数据类型所提供的操作使用 Java 语言给出明确的定义。事实上对抽象数据类型提供的操作使用高级语言进行定义，就是给出其应用程序接口，在 Java 中我们可以使用一个接口来进行定义。如此在使用类来完成抽象数据类型的具体实现时，我们只要实现相应的接口就实现了对抽象数据类型定义的操作的实现。

为此我们给出抽象数据类型 List 的 Java 接口。

代码 3-1 List 接口

```
public interface List {
    //返回线性表的大小，即数据元素的个数。
    public int getSize();
    //如果线性表为空返回 true，否则返回 false。
    public boolean isEmpty();
    //判断线性表是否包含数据元素 e
    public boolean contains(Object e);
    //返回数据元素 e 在线性表中的序号
    public int indexOf(Object e);
    //将数据元素 e 插入到线性表中 i 号位置
    public void insert(int i, Object e) throws OutOfBoundaryException;
    //将数据元素 e 插入到元素 obj 之前
    public boolean insertBefore(Object obj, Object e);
    //将数据元素 e 插入到元素 obj 之后
    public boolean insertAfter(Object obj, Object e);
    //删除线性表中序号为 i 的元素,并返回之
    public Object remove(int i) throws OutOfBoundaryException;
    //删除线性表中第一个与 e 相同的元素
    public boolean remove(Object e);
    //替换线性表中序号为 i 的数据元素为 e，返回原数据元素
    public Object replace(int i, Object e) throws OutOfBoundaryException;
    //返回线性表中序号为 i 的数据元素
    public Object get(int i) throws OutOfBoundaryException;
}
```

在上述 List 接口的定义中我们将数据元素的类型定义为 Object 类型，做这样的定义的原因是：在 Java 中 Object 类是其他所有类的父类，因此其他任何类的引用或者说任何类类型的变量都可以赋给 Object 类型的变量，这样我们实现的抽象数据类型就可以对任何一种

数据元素都适用，而不用对每一种不同类型的数据元素给出不同的实现。也就是说我们将要实现的线性表可以存放任何一种数据元素。

其次，在 List 接口的定义中使用了异常。这是因为在某些操作的实现过程中，会出现各种错误的情况。这些错误可能是用户的要求无法实现，例如线性表已经为空，但是用户仍然调用删除数据元素的方法，那么此时删除操作是无法实现的；又或者用户在使用这些操作时出现了错误，例如线性表虽然不为空，但是用户在调用 get(int i)方法时指定的序号 i 超过了范围，此时也会出错。因此在定义接口时还要对各种可能出现的错误条件，定义相应的异常。异常的定义可以通过继承 java.lang.Exception 或其子类来实现。

代码 3-2 OutOfBoundaryException 异常

```
//线性表中出现序号越界时抛出该异常
public class OutOfBoundaryException extends RuntimeException{
    public OutOfBoundaryException(String err){
        super(err);
    }
}
```

3.1.4 Strategy 接口

在 List 接口方法定义中，我们将所有数据元素的类型都使用 Object 来替代，这样做是为了程序的通用性，即一种抽象数据类型的实现可以用于所有数据元素。但是这样做带来了另一个需要解决的问题，即完成数据元素之间比较大小或是否相等的问题。在使用 Object 类型的变量指代了所有数据类型之后，那么所有种类的数据元素的比较就都需要使用 Object 类型的变量来完成，但是不同数据元素的比较方法或策略是不一样的。例如字符串的比较是使用 java.lang.String 类的 compareTo 和 equals 方法；而基本的数值型数据是使用关系运算符来完成的；其他各种不同的类的比较方法就更加千差万别多种多样了，即使同一个类的比较方法在不同的情况下也会不同，例如两个学生之间的比较有时可以用学号的字典顺序来进行，有时可能又需要使用成绩来比较。因此我们无法简单的在两个 Object 类型的变量之间使用"= ="、"<"等关系运算符来完成各种不同数据元素之间的比较操作，同时 Java 也不提供运算符的重载，为此我们引入 Strategy 接口。

使用 Strategy 接口可以实现各种不同数据元素相互之间独立的比较策略。在实现各种抽象数据类型时，例如线性表，可以使用 Strategy 接口变量来完成形式上的比较，然后在创建每个抽象数据类型的实例时，例如一个具体的用于学生的线性表时，可以引入一个实际的实现了 Strategy 接口的策略类对象，例如实现了 Strategy 接口的学生比较策略类对象。使用这一策略的另一优点在于，一旦不想继续使用原先的比较策略对象，随时可以使用另一个比较策略对象将其替换，而不同修改抽象数据类型的具体实现。

按上述方案我们给出相应的代码。

代码 3-3 Strategy 接口

```
public interface Strategy {
    //判断两个数据元素是否相等
    public boolean equal(Object obj1, Object obj2);
    /**
     * 比较两个数据元素的大小
     * 如果 obj1 < obj2  返回-1
     * 如果 obj1 = obj2  返回 0
```

```
     * 如果 obj1 > obj2 返回 1
     */
    public int compare(Object obj1, Object obj2);
}
```

例如对于学生可以给出以下比较策略。

代码 3-4 学生比较策略

```
public class StudentStrategy implements Strategy {
    public boolean equal(Object obj1, Object obj2) {
        if (obj1 instanceof Student && obj2 instanceof Student) {
            Student s1 = (Student)obj1;
            Student s2 = (Student)obj2;
            return s1.getSId().equals(s2.getSId());
        }
        else
            return false;
    }
    public int compare(Object obj1, Object obj2) {
        if (obj1 instanceof Student && obj2 instanceof Student) {
            Student s1 = (Student)obj1;
            Student s2 = (Student)obj2;
            return s1.getSId().compareTo(s2.getSId());
        } else
            return obj1.toString().compareTo(obj2.toString());
    }
}
```

3.2 线性表的顺序存储与实现

线性表的顺序存储是用一组地址连续的存储单元依次存储线性表的数据元素。假设线性表的每个数据元素需占用K个存储单元，并以元素所占的第一个存储单元的地址作为数据元素的存储地址。则线性表中序号为i的数据元素的存储地址LOC(a_i)与序号为i+1 的数据元素的存储地址LOC(a_{i+1})之间的关系为

$$LOC(a_{i+1}) = LOC(a_i) + K$$

通常来说，线性表的i号元素a_i的存储地址为

$$LOC(a_i) = LOC(a_0) + i \times K$$

其中LOC(a_0)为 0 号元素a_0的存储地址，通常称为线性表的起始地址。

线性表的这种机内表示称作线性表的顺序存储。它的特点是，以数据元素在机内存储地址相邻来表示线性表中数据元素之间的逻辑关系。每一个数据元素的存储地址都和线性表的起始地址相差一个与数据元素在线性表中的序号成正比的常数。由此，只要确定了线性表的起始地址，线性表中的任何一个数据元素都可以随机存取，因此线性表的顺序存储结构是一种随机的存储结构。线性表的顺序存储可用图 3-1 描述。

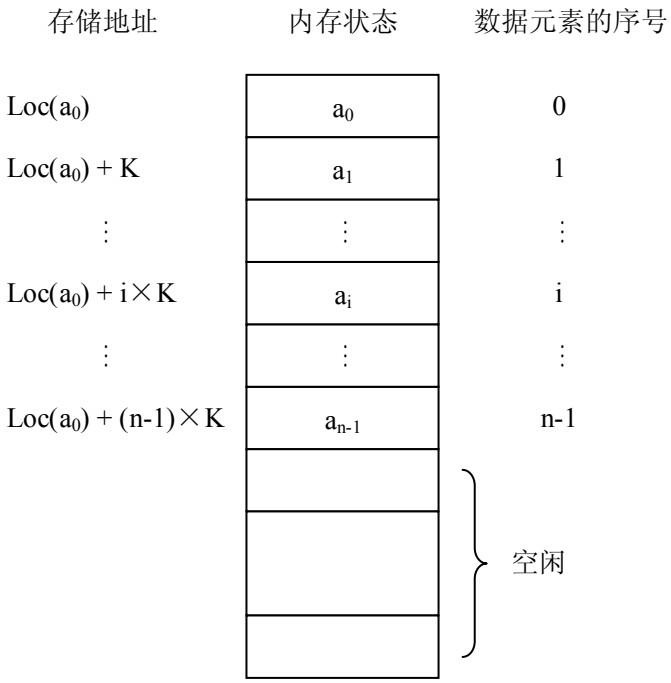


图 3-1 线性表的顺序存储

由于高级语言中的数组具也有随机存储的特性，因此在抽象数据类型的实现中都是使用数组来描述数据结构的顺序存储结构。通过图 3-1，我们看到线性表中的数据元素在依次存放到数组中的时候，线性表中序号为 i 的数据元素对应的数组下标也为 i，即数据元素在线性表中的序号与数据元素在数组中的下标相同。

在这里需要注意的是，如果线性表中的数据元素是对象时，数组存放的是对象的引用，即线性表中所有数据元素的对象引用是存放在一组连续的地址空间中。如图 3-2 所示。

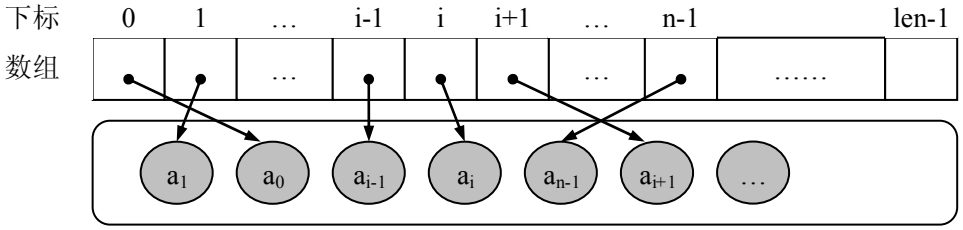


图 3-2 数组存储对象引用

由于线性表的长度可变，不同的问题所需的最大长度不同，那么在线性表的具体实现中我们是使用动态扩展数组大小的方式来完成线性表长度的不同要求的。在第二章的例 2-11 中我们看到了使用动态扩展数组大小的方式来实现这一存储策略的方法，以及动态扩展数组方法的时间复杂度，对于每一个元素而言其均摊时间复杂度为 $\Theta(1)$。因此在分析算法时间复杂度时，使用动态扩展数组只会给算法增加常数运行时间。

在使用数组实现线性表的操作中，经常会碰到在数组中进行数据元素的查找、添加、删除等操作，下面我们先讨论如何在数组中实现上述操作。

在数组中进行查找，最简单的方法就是算法 2-5 描述的顺序查找，其平均时间复杂度是 $\Theta(n)$。这些在算法 2-5 及例 2-10 中已经详细分析，在这里不再赘述。

在数组中添加数据元素，通常是在数组中下标为 i $(0 \le i \le n)$的位置添加数据元素，而将原来下标从 i 开始的数组中所有后续元素依次后移。整个操作过程可以通过图 3-3 说明。

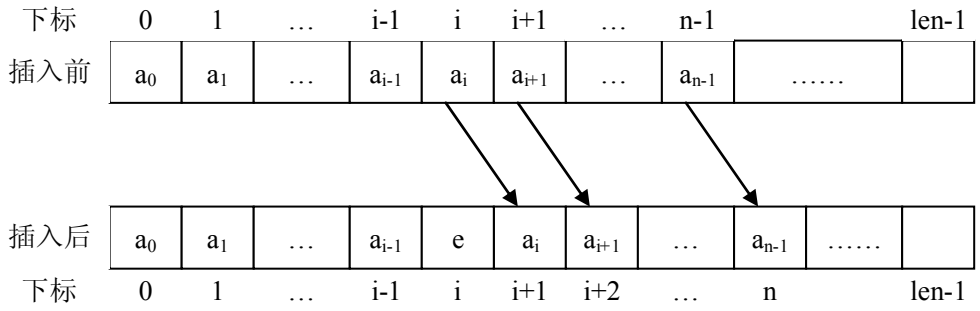


图 3-3 在数组下标 i 处插入元素 e

使用 Java 语言实现整个操作过程的关键语句是

```
for (int j=n; j>i; j--)
    a[j] = a[j-1];
a[i] = e;
```

如果要对上述操作的执行时间进行分析，那么对于不同下标处的插入，情况会有所不同。我们主要关心平均情况下的运行时间，并假设在数组下标[0, n]范围内任何一个位置 i 处插入数据元素的概率是相等的。即

$$\forall k \in [0,n], P[i=k] = \frac{1}{n+1}$$

假设C_i是在数组下标为i的地方插入数据元素时需要移动数据元素的次数，那么Ci = n - i， $0 \le i \le n$。此时，在数组下标i处插入一个数据元素需要移动数据元素的平均次数是：

$$T(n) = \sum_{i=0}^{n}(Ci \cdot P[i]) = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n}{2} = \Theta(n)$$

因此，在一个具有 n 个数据元素的数组中插入一个数据元素的平均时间复杂度为 Θ(n)。

与在数组中添加数据元素相对的是在数组中删除数据元素，与添加类似，删除操作也通常是删除数组中下标为 i ($0 \le i < n$)的元素，然后将数组中下标从 i+1 开始的所有后续元素依次前移。删除操作过程也可以通过图 3-4 说明。

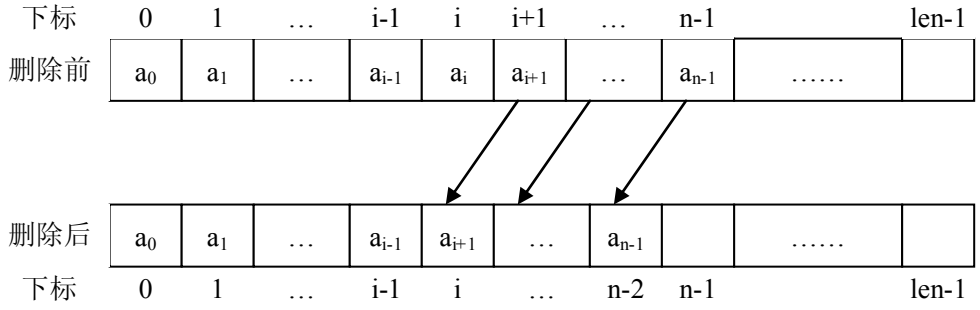


图 3-4 在数组下标 i 处删除元素

使用 Java 语言实现整个操作过程的关键语句是

```
        for (int j=i; j<n-1; j++)
            a[j] = a[j+1];
```

同样在对删除操作的执行时间进行分析时，我们也关心平均情况下的运行时间，假设在数组下标[0, n-1]范围内任何一个位置 i 处删除数据元素的概率是相等的。即

$$\forall k \in [0, n-1], P[i = k] = \frac{1}{n}$$

假设C_i是在数组下标为i的地方删除数据元素时需要移动数据元素的次数，那么Ci = n -1 - i，$0 \le i < n$。此时，在数组下标i处删除一个数据元素需要移动数据元素的平均次数是：

$$T(n) = \sum_{i=0}^{n-1} (Ci \cdot P[i]) = \frac{1}{n} \sum_{i=0}^{n-1} (n-1-i) = \frac{1}{n} \frac{n(n-1)}{2} = \frac{n-1}{2} = \Theta(n)$$

因此，在一个具有 n 个数据元素的数组中删除一个数据元素的平均时间复杂度为 Θ(n)。

在对数组中的相关操作进行讨论后，下面我们给出使用数组实现线性表的代码。

代码 3-5 线性表的数组实现

```
public class ListArray implements List {
    private final int LEN = 8;      //数组的默认大小
    private Strategy strategy;      //数据元素比较策略
    private int size;               //线性表中数据元素的个数
    private Object[] elements;      //数据元素数组
    //构造方法
    public ListArray () {
        this(new DefaultStrategy());
    }
    public ListArray (Strategy strategy){
        this.strategy = strategy;
        size = 0;
        elements = new Object[LEN];
    }

    //返回线性表的大小，即数据元素的个数。
    public int getSize() {
        return size;
    }

    //如果线性表为空返回 true，否则返回 false。
    public boolean isEmpty() {
        return size==0;
    }

    //判断线性表是否包含数据元素 e
    public boolean contains(Object e) {
        for (int i=0; i<size; i++)
```

```
            if (strategy.equal(e,elements[i])) return true;
        return false;
    }

    //返回数据元素 e 在线性表中的序号
    public int indexOf(Object e) {
        for (int i=0; i<size; i++)
            if (strategy.equal(e,elements[i])) return i;
        return -1;
    }

    //将数据元素 e 插入到线性表中 i 号位置
    public void insert(int i, Object e) throws OutOfBoundaryException {
        if (i<0||i>size)
            throw new OutOfBoundaryException("错误，指定的插入序号越界。");
        if (size >= elements.length)
            expandSpace();
        for (int j=size; j>i; j--)
            elements[j] = elements[j-1];
        elements[i] = e;       size++;
        return;
    }

    private void expandSpace(){
        Object[] a = new Object[elements.length*2];
        for (int i=0; i<elements.length; i++)
            a[i] = elements[i];
        elements = a;
    }

    //将数据元素 e 插入到元素 obj 之前
    public boolean insertBefore(Object obj, Object e) {
        int i = indexOf(obj);
        if (i<0) return false;
            insert(i,e);
        return true;
    }

    //将数据元素 e 插入到元素 obj 之后
    public boolean insertAfter(Object obj, Object e) {
        int i = indexOf(obj);
        if (i<0) return false;
            insert(i+1,e);
        return true;
```

```
    }
    //删除线性表中序号为 i 的元素,并返回之
    public Object remove(int i) throws OutOfBoundaryException {
        if (i<0||i>=size)
            throw new OutOfBoundaryException("错误，指定的删除序号越界。");
        Object obj = elements[i];
        for (int j=i; j<size-1; j++)
            elements[j] = elements[j+1];
        elements[--size] = null;
        return obj;
    }

    //删除线性表中第一个与 e 相同的元素
    public boolean remove(Object e) {
        int i = indexOf(e);
        if (i<0) return false;
        remove(i);
        return true;
    }

    //替换线性表中序号为 i 的数据元素为 e，返回原数据元素
    public Object replace(int i, Object e) throws OutOfBoundaryException {
        if (i<0||i>=size)
            throw new OutOfBoundaryException("错误，指定的序号越界。");
        Object obj = elements[i];
        elements[i] = e;
        return obj;
    }

    //返回线性表中序号为 i 的数据元素
    public Object get(int i) throws OutOfBoundaryException {
        if (i<0||i>=size)
            throw new OutOfBoundaryException("错误，指定的序号越界。");
        return elements[i];
    }
}
```

代码 3-5 说明：在 ArrayList 类中共有 4 个成员变量，其中 elements 数组以及 size 用于存储线性表中的数据元素以及表明线性表中数据元素的个数；而 strategy 是用来完成线性表中数据元素的比较操作的策略；LEN 是 elements 数组的初始默认大小，数组的大小在后续的插入操作中可以会发生变化。

算法 getSize()、isEmpty()、replace(int i, Object e)、get(int i)的时间复杂度均为 $\Theta(1)$。通过成员变量 size 可以直接判断出线性表中数据元素的个数以及线性表是否为空。这里使用数组来实现线性表，由于数组具有随机存取的特性，因此获取线性表中序号为 i 的数据元素或对其进行替换均可在常数时间内完成。

算法contains(Object e)、indexOf(Object e)主要是在线性表中查找某个数据元素，它们与算法 2-5 linearSearch类似，只是存在查找可能会出现不成功的情况。此时可以假设在具有n个数据元素的线性表中包含一个本不属于线性表的数据元素a_{n+1}，如果把查找不成功的情况对应为查找本不属于线性表的数据元素a_{n+1}，则上述两个算法数组实现的平均时间复杂度可以对应为在具有n+1 个数据元素的数组中查找成功的情况（这时查找不成功的机率为1/(n+1)），算法运行时间T(n)= ((n+1)+1)/2 ≈ n/2。即完成上述操作需要比较数组中近一半的元素。

算法 insert(int i, Object e)、remove(int i)主要是按照线性表中的序号来完成数据元素的插入与删除。在使用数组实现时，算法的时间复杂度在对数组基本操作的分析中已经说明，要完成这些操作平均要移动数组中大约一半的数据元素。并且如果在插入数据元素的过程中，出现了数组空间的扩展，通过前面的均摊分析我们知道对于每个数据元素而言需要的时间是常数，因此算法的运行时间 T(n) ≈n/2。

算法 insertBefore(Object obj, Object e)、insertAfter(Object obj, Object e)、remove(Object e)是按照线性表中的某个特定数据元素来完成数据元素的插入、删除操作。此时算法可以看成由两个部分组成，首先需要在线性表中找到对应的数据元素，然后按照数据元素在线性表中的位置来完成相应的插入和删除操作。在使用数组实现时，算法的运行时间 T(n) ≈ n。下面以算法 insertBefore 为例来说明：如果 p 不存在于数组中，则整个算法需要进行 n 次比较 0 次移动；如果 p 存在于数组中，并且对应的数组下标为 i，则第一步需要进行 i+1 次比较才能找到 p，第二步需要依次后移从 i 开始的 n-i 个数据元素，因此整个算法的运行时间 T(n) = (i+1) + (n-i) = n+1 ≈ n。同样在插入的过程中如果出现了数组空间的扩展，对于每个元素而言只会增加常数时间，不会对算法的运行时间造成实质性的影响。

3.3 线性表的链式存储与实现

实现线性表的另一种方法是链式存储，即用指针将存储线性表中数据元素的那些单元依次串联在一起。这种方法避免了在数组中用连续的单元存储元素的缺点，因而在执行插入或删除运算时，不再需要移动元素来腾出空间或填补空缺。然而我们为此付出的代价是，需要在每个单元中设置指针来表示表中元素之间的逻辑关系，因而增加了额外的存储空间的开销。

上述实现方法实际上就是使用链表来实现线性表。而链表有许多不同的形式，在这一节中首先介绍两种重要的链表及其操作特点，然后给出一种线性表的链表实现。

3.3.1 单链表

链表是一系列的存储数据元素的单元通过指针串接起来形成的，因此每个单元至少有两个域，一个域用于数据元素的存储，另一个域是指向其他单元的指针。这里具有一个数据域和多个指针域的存储单元通常称为**结点（node）**。

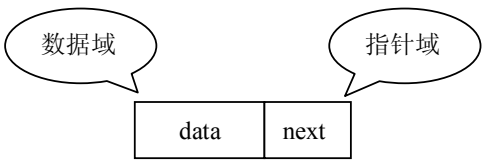


图 3-5 单链表结点结构

一种最简单的结点结构如图 3-5 所示，它是构成单链表的基本结点结构。在结点中数据域用来存储数据元素，指针域用于指向下一个具有相同结构的结点。

在 Java 中没有显式的指针类型，然而实际上对象的访问就是使用指针来实现的，即在 Java 中是使用对象的引用来替代指针的。因此在使用 Java 实现该结点结构时，一个结点本身就是一个对象。结点的数据域 data 可以使用一个 Object 类型的对象来实现，用于存储任何类型的数据元素，并通过对象的引用指向该元素；而指针域 next 可以通过节点对象的引用来实现。

由于数据域存储的也是对象引用，因此数据实际上和图 3-2 中一样，是通过指向数据的物理存储地址来完成存储的，但是在后面叙述的方便，我们在图示中都将数据元素直接画到了数据域中，请读者注意实际的状态与之是有区别的。

上面的单链表结点结构是结点的一种最简单的形式，除此之外还有其他不同的结点结构，但是这些结点结构都有一个数据域，并均能完成数据元素的存取。为此在使用 Java 定义单链表结点结构之前先给出一个结点接口，在接口中定义了所有结点均支持的操作，即对结点中存储数据的存取。代码 3-6 定义了结点接口。

代码 3-6 结点接口

```
public interface Node {
    //获取结点数据域
    public Object getData();
    //设置结点数据域
    public void setData(Object obj);
}
```

在给出结点接口定义之后，单链表的结点定义就可以通过实现结点接口来完成。代码 3-7 给出了单链表结点的定义。

代码 3-7 单链表结点定义

```
public class SLNode implements Node {
    private Object element;
    private SLNode next;

    public SLNode() {
        this(null,null);
    }
    public SLNode(Object ele, SLNode next){
        this.element = ele;
        this.next = next;
    }

    public SLNode getNext(){
        return next;
    }
    public void setNext(SLNode next){
        this.next = next;
    }
    /**************** Methods of Node Interface **************/
    public Object getData() {
```

```
        return element;
    }
    public void setData(Object obj) {
        element = obj;
    }
}
```

单链表是通过上述定义的结点使用 next 域依次串联在一起而形成的。一个单链表的结构如图 3-6 所示。

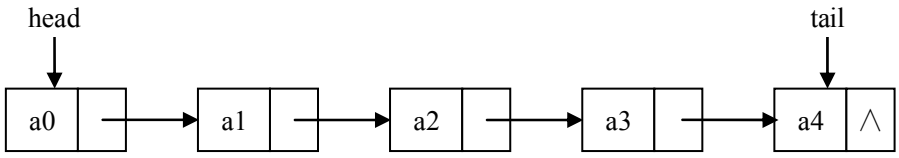


图 3-6 单链表结构

链表的第一个结点和最后一个结点，分别称为链表的**首结点**和**尾结点**。尾结点的特征是其 next 引用为空（null）。链表中每个结点的 next 引用都相当于一个指针，指向另一个结点，借助这些 next 引用，我们可以从链表的首结点移动到尾结点。如此定义的结点称为**单链表（single linked list）**。在单链表中通常使用 head 引用来指向链表的首结点，由 head 引用可以完成对整个链表中所有节点的访问。有时也可以根据需要使用指向尾结点的 tail 引用来方便某些操作的实现。

在单链表结构中还需要注意的一点是，由于每个结点的数据域都是一个 Object 类的对象，因此，每个数据元素并非真正如图 3-4 中那样，而是在结点中的数据域通过一个 Object 类的对象引用来指向数据元素的。

与数组类似，单链表中的结点也具有一个线性次序，即如果结点 P 的 next 引用指向结点 S，则 P 就是 S 的直接前驱，S 是 P 的直接后续。单链表的一个重要特性就是只能通过前驱结点找到后续结点，而无法从后续结点找到前驱结点。在单链表中通常需要完成数据元素的查找、插入、删除等操作。下面我们逐一讨论这些操作的实现。

在单链表中进行查找操作，只能从链表的首结点开始，通过每个结点的 next 引用来一次访问链表中的每个结点以完成相应的查找操作。例如需要在单链表中查找是否包含某个数据元素 e，则方法是使用一个循环变量 p，起始时从单链表的头结点开始，每次循环判断 p 所指结点的数据域是否和 e 相同，如果相同则可以返回 true，否则继续循环直到链表中所有结点均被访问，此时 p 为 null。该过程如图 3-7 所示。

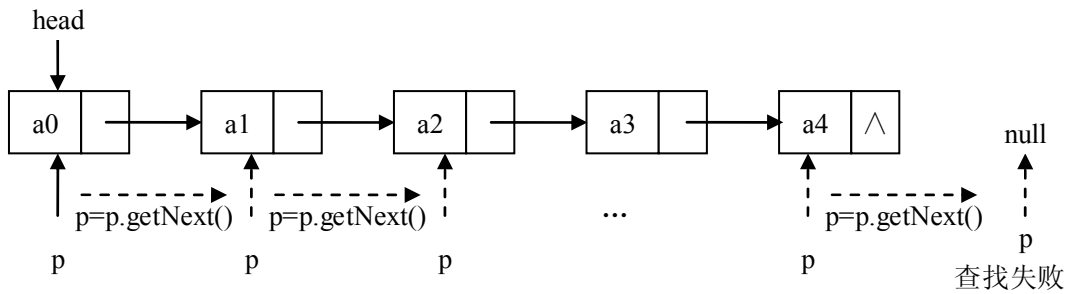


图 3-7 在单链表中查找元素

使用 Java 语言实现整个过程的关键语句是：

```
p=head;
while (p!=null)
      if (strategy.equal( e , p.getData() )) return true;
return false;
```

在单链表中查找操作的运行时间与在数组中的查找操作一样，在平均情况下需要比较大约一般的数据元素，即 T(n) ≈ n/2。

在单链表中数据元素的插入，是通过在链表中插入数据元素所属的结点来完成的。对于链表的不同位置，插入的过程会有细微的差别。图 3-8（a）、3-8（b）、3-8（c）分别说明了在单链表的表头、表尾以及链表中间插入结点的过程。

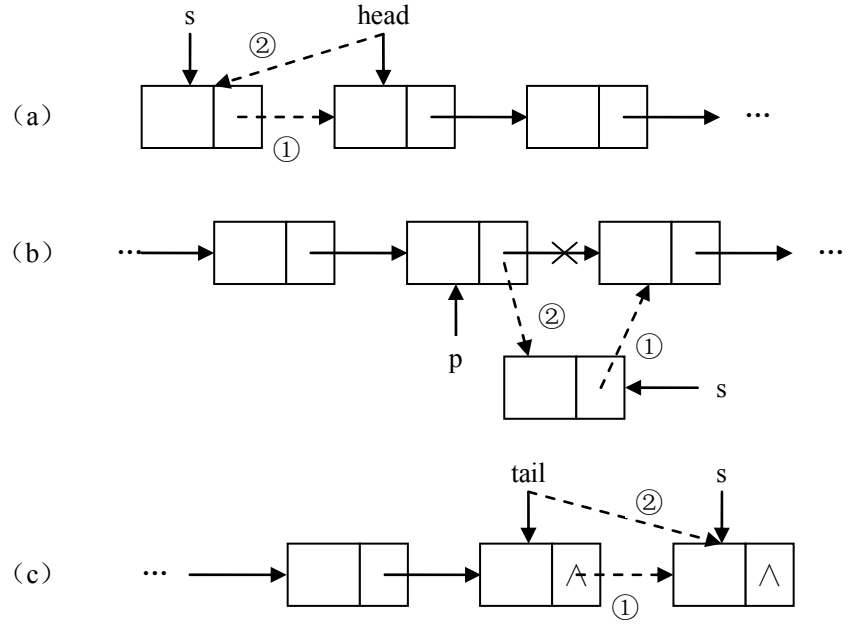
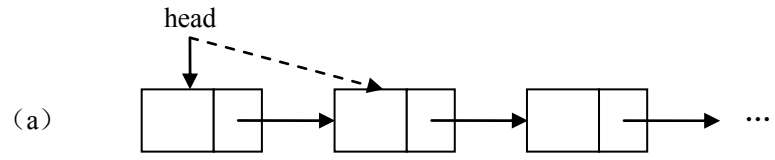


图 3-8 在单链表中插入结点

从图 3-7 中可以看出，除了单链表的首结点由于没有直接前驱结点，所以可以直接在首结点之前插入一个新的结点之外，在单链表中的其他任何位置插入一个新结点时，都只能是在已知某个特定结点引用的基础上在其后面插入一个新结点。并且在已知单链表中某个结点引用的基础上，完成结点的插入操作需要的时间是 Θ(1)。由于在单链表中数据元素的插入是通过节点的插入来完成的，因此在单链表中完成数据元素的插入操作要比在数组中完成数据元素的插入操作所需 O(n)的时间要快得多。

类似的，在单链表中数据元素的删除也是通过结点的删除来完成的。在链表的不同位置删除结点，其操作过程也会有一些差别。图 3-9（a）、3-9（b）、3-9（c）分别说明了在单链表的表头、表尾以及链表中间删除结点的过程。



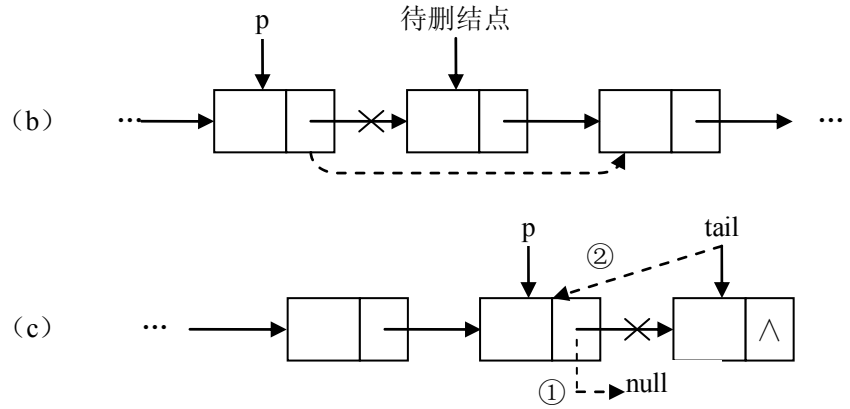


图 3-9 在单链表中插入结点

从图 3-8 中可以看出，在单链表中删除一个结点时，除首结点外都必须知道该结点的直接前驱结点的引用。并且在已知单链表中某个结点引用的基础上，完成其后续结点的删除操作需要的时间是Θ(1)。由于在单链表中数据元素的删除是通过节点的删除来完成的，因此在单链表中完成数据元素的删除操作要比在数组中完成数据元素的删除操作所需O(n)的时间要快得多。

通过以上分析，我们可以得出以下结论：在单链表中进行顺序查找与在数组中完成相同操作具有相同的时间复杂度，而在单链表中在已知特定结点引用的前提下完成数据元素的插入与删除操作要比在数组中完成相同操作快得多。

3.3.2 双向链表

单链表的一个优点是结构简单，但是它也有一个缺点，即在单链表中只能通过一个结点的引用访问其后续结点，而无法直接访问其前驱结点，要在单链表中找到某个结点的前驱结点，必须从链表的首结点出发依次向后寻找，但是需要O(n)时间。为此我们可以扩展单链表的结点结构，使得通过一个结点的引用，不但能够访问其后续结点，也可以方便的访问其前驱结点。扩展单链表结点结构的方法是，在单链表结点结构中新增加一个域，该域用于指向结点的直接前驱结点。扩展后的结点结构是构成双向链表的结点结构，如图 3-10 所示。

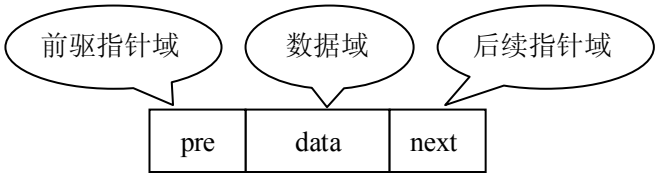


图 3-10 双向链表结点结构

与单链表节点定义类似，双向链表的结点定义也可以通过实现结点接口来完成。代码 3-8 给出了双向链表结点的定义。

代码 3-8 双向链表结点定义

```
public class DLNode implements Node {
    private Object element;
    private DLNode pre;
    private DLNode next;
```

```
    public DLNode() {
        this(null,null,null);
    }
    public DLNode(Object ele, DLNode pre, DLNode next){
        this.element = ele;
        this.pre = pre;
        this.next = next;
    }

    public DLNode getNext(){
        return next;
    }
    public void setNext(DLNode next){
        this.next = next;
    }
    public DLNode getPre(){
        return pre;
    }
    public void setPre(DLNode pre){
        this.pre = pre;
    }
    /****************Node Interface Method*************/
    public Object getData() {
        return element;
    }
    public void setData(Object obj) {
        element = obj;
    }
}
```

双向链表是通过上述定义的结点使用 pre 以及 next 域依次串联在一起而形成的。一个双向链表的结构如图 3-11 所示。

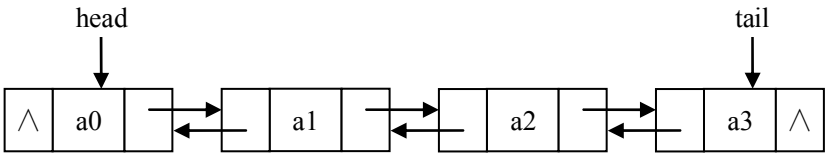


图 3-11 双向链表结构

在双向链表中同样需要完成数据元素的查找、插入、删除等操作。在双向链表中进行查找与在单链表中类似，只不过在双向链表中查找操作可以从链表的首结点开始，也可以从尾结点开始，但是需要的时间和在单链表中一样，在平均情况下需要比较大约一般的数据元素，即 T(n) ≈ n/2。

单链表的插入操作，除了首结点之外必须在某个已知结点后面进行，而在双向链表中插入操作在一个已知的结点之前或之后都可以进行。例如在某个结点 p 之前插入一个新结点的

过程如图 3-12 所示。

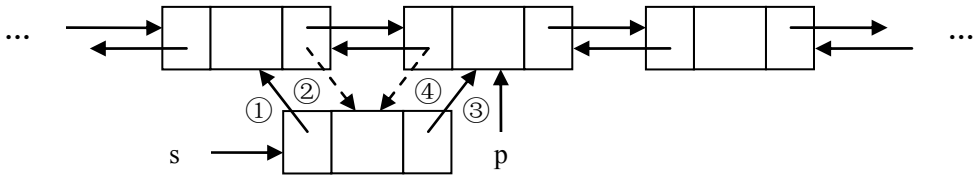


图 3-12 在结点 p 之前插入 s

使用 Java 语言实现整个过程的关键语句是

```
s.setPre (p.getPre());
p.getPre().setNext(s);
s.setNext(p);
p.setPre(s);
```

在结点 p 之后插入一个新结点的操作与上述操作对称，这里不再赘述。插入操作除了上述情况，还可以在双向链表的首结点之前、双向链表的尾结点之后进行，此时插入操作与上述插入操作相比更为简单，请读者自己分析。

单链表的删除操作，除了首结点之外必须在知道待删结点的前驱结点的基础上才能进行，而在双向链表中在已知某个结点引用的前提下，可以完成该结点自身的删除。图 3-13 表示了删除 p 的过程。

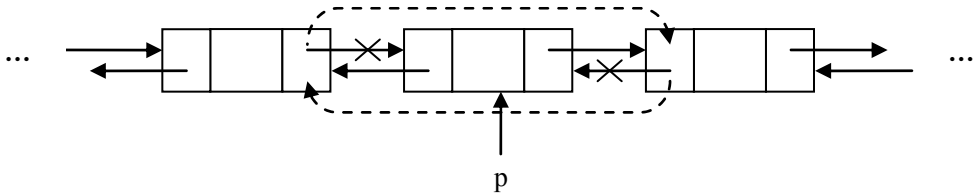


图 3-13 删除结点 p

使用 Java 语言实现整个过程的关键语句是

```
p.getPre().setNext(p.getNext());
p.getNext().setPre(p.getPre());
```

如果删除的结点是首结点或尾结点时，情况会更加简单，请读者自己分析。

3.3.3 线性表的单链表实现

在使用链表实现线性表时，既可以使用单链表，也可以使用双向链表。实现中链表的选择主要是依据需要实现的ADT的基本操作来决定，在这里我们可以选择单链表来实现线性表。在使用单链表实现线性表时，线性表中的每个数据元素对应单链表中的一个结点，而线性表元素之间的逻辑关系是通过单链表中元素所在结点之间的指向来表示的：如果表是a_0, a_1, …, a_{n-1} ，那么含有元素a_{i-1}的结点的next域应指向含有元素a_i的结点(i=1,2,…,n-1)。含有a_{n-1}的那个结点的next域是null。

在使用单链表实现线性表的时候，为了使程序更加简洁，我们通常在单链表的最前面添加一个哑元结点，也称为头结点。在头结点中不存储任何实质的数据对象，其 next 域指向线性表中 0 号元素所在的结点，头结点的引入可以使线性表运算中的一些边界条件更容易处

理。一个带头结点的单链表实现线性表的结构图如图 3-14 所示。

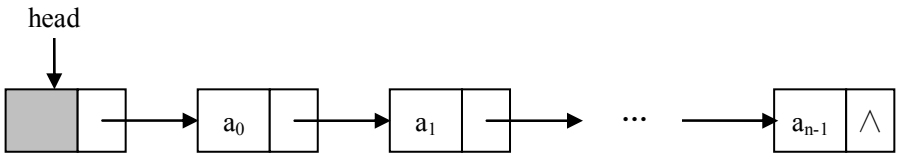


图 3-14 带头结点的单链表

通过图 3-12 我们发现，对于任何基于序号的插入、删除，以及任何基于数据元素所在结点的前面或后面的插入、删除，在带头结点的单链表中均可转化为在某个特定结点之后完成结点的插入、删除，而不用考虑插入、删除是在链表的首部、中间、还是尾部等不同情况。代码 3-9 给出了基于单链表实现线性表的程序。

代码 3-9 线性表的单链表实现

```
public class ListSLinked implements List {
    private Strategy strategy;      //数据元素比较策略
    private SLNode head;            //单链表首结点引用
    private int size;               //线性表中数据元素的个数

    public ListSLinked () {
        this(new DefaultStrategy());
    }
    public ListSLinked (Strategy strategy) {
        this.strategy = strategy;
        head = new SLNode();
        size = 0;
    }

    //辅助方法：获取数据元素 e 所在结点的前驱结点
    private SLNode getPreNode(Object e){
        SLNode p = head;
        while (p.getNext()!=null)
            if (strategy.equal(p.getNext().getData(),e)) return p;
            else  p = p.getNext();
        return null;
    }
    //辅助方法：获取序号为 0<=i<size 的元素所在结点的前驱结点
    private SLNode getPreNode(int i){
        SLNode p = head;
        for (; i>0; i--)  p = p.getNext();
        return p;
    }
    //获取序号为 0<=i<size 的元素所在结点
    private SLNode getNode(int i){
        SLNode p = head.getNext();
```

```
        for (; i>0; i--)  p = p.getNext();
        return p;
    }

    //返回线性表的大小，即数据元素的个数。
    public int getSize() {
        return size;
    }

    //如果线性表为空返回 true，否则返回 false。
    public boolean isEmpty() {
        return size==0;
    }

    //判断线性表是否包含数据元素 e
    public boolean contains(Object e) {
        SLNode p = head.getNext();
        while (p!=null)
            if (strategy.equal(p.getData(),e)) return true;
            else  p = p.getNext();
        return false;
    }

    //返回数据元素 e 在线性表中的序号
    public int indexOf(Object e) {
        SLNode p = head.getNext();
        int index = 0;
        while (p!=null)
            if (strategy.equal(p.getData(),e)) return index;
            else  {index++; p = p.getNext();}
        return -1;
    }

    //将数据元素 e 插入到线性表中 i 号位置
    public void insert(int i, Object e) throws OutOfBoundaryException {
        if (i<0||i>size)
            throw new OutOfBoundaryException("错误，指定的插入序号越界。");
        SLNode p = getPreNode(i);
        SLNode q = new SLNode(e,p.getNext());
        p.setNext(q);
        size++;
        return;
    }
```

```
//将数据元素 e 插入到元素 obj 之前
public boolean insertBefore(Object obj, Object e) {
    SLNode p = getPreNode(obj);
    if (p!=null){
        SLNode q = new SLNode(e,p.getNext());
        p.setNext(q);
        size++;
        return true;
    }
    return false;
}

//将数据元素 e 插入到元素 obj 之后
public boolean insertAfter(Object obj, Object e) {
    SLNode p = head.getNext();
    while (p!=null)
        if (strategy.equal(p.getData(),obj)){
            SLNode q = new SLNode(e,p.getNext());
            p.setNext(q);
            size++;
            return true;
        }
        else  p = p.getNext();
    return false;
}

//删除线性表中序号为 i 的元素,并返回之
public Object remove(int i) throws OutOfBoundaryException {
    if (i<0||i>=size)
        throw new OutOfBoundaryException("错误，指定的删除序号越界。");
    SLNode p = getPreNode(i);
    Object obj = p.getNext().getData();
    p.setNext(p.getNext().getNext());
    size--;
    return obj;
}

//删除线性表中第一个与 e 相同的元素
public boolean remove(Object e) {
    SLNode p = getPreNode(e);
    if (p!=null){
        p.setNext(p.getNext().getNext());
        size--;
        return true;
```

```
        }
        return false;
    }

    //替换线性表中序号为 i 的数据元素为 e，返回原数据元素
    public Object replace(int i, Object e) throws OutOfBoundaryException {
        if (i<0||i>=size)
            throw new OutOfBoundaryException("错误，指定的序号越界。");
        SLNode p = getNode(i);
        Object obj = p.getData();
        p.setData(e);
        return obj;
    }

    //返回线性表中序号为 i 的数据元素
    public Object get(int i) throws OutOfBoundaryException {
        if (i<0||i>=size)
            throw new OutOfBoundaryException("错误，指定的序号越界。");
        SLNode p = getNode(i);
        return p.getData();
    }
}
```

代码 3-9 说明：在 SLinkedList 类中共有 3 个成员变量，其中 size 用于表明线性表中数据元素的个数；head 是带头结点的单链表的首结点引用；而 strategy 是用来完成线性表中数据元素的比较操作的策略。

算法 getSize()、isEmpty()的时间复杂度均为 $\Theta(1)$。通过成员变量 size 可以直接判断出线性表中数据元素的个数以及线性表是否为空。

在类中提供了两个私有方法 getPreNode(Object e)、getPreNode(int i)，其功能是找到数据元素 e 或线性表中 i 号数据元素所在结点的前驱结点。在带头结点的单链表中的插入、删除操作均是在某个结点之后完成的，因此线性表中一些基于数据元素或序号的插入、删除操作的实现依赖于对应元素在单链表中的前驱结点引用。这两个方法的平均运行时间 $T(n)\approx n/2$。

算法 replace(int i, Object e)、get(int i)的平均时间复杂度均为 $\Theta(n)$。由于链表中每个结点在内存中的地址并不是连续的，所以链表不具有随机存取的特性，这样要对线性表中 i 号元素进行获取或替换的操作，不可能与使用数组实现线性表那样可以在常数时间内完成，而是必须从链表的头结点开始沿着链表定位 i 号元素所在的结点，然后才能进行相应的操作，因此算法的平均运行时间 $T(n)\approx n/2$，比使用数组实现相应操作要慢得多。

算法 contains(Object e)、indexOf(Object e)主要是在线性表中查找某个数据元素。算法平均运行时间与使用数组的实现一样，都需要从线性表中 0 号元素出发，依次向后查找，因此算法运行时间 $T(n) \approx n/2$。

算法 insert(int i, Object e)、remove(int i)在实现的过程中首先需要在链表中定位 i 号元素所在结点的前驱结点，然后才能完成插入、删除操作，由于定位方法 getPreNode(Object e)、getPreNode(int i)的平均运行时间约为 n/2，而真正的结点的插入与删除只需要常数时间，因此算法的运行时间 $T(n)\approx n/2$，与使用数组实现的运行时间相同。

算法 insertBefore(Object obj, Object e)、insertAfter(Object obj, Object e)、remove(Object e)在实现的过程中 insertBefore、remove 需要找到对应元素的前驱结点，insertAfter 需要找到对应元素本身，这个定位过程的平均运行时间约为 n/2，而剩下的插入与删除操作只需要常数时间，因此整个算法的平均运行时间 $T(n) \approx n/2 < n$，要优于使用数组实现的运行时间。

3.4 两种实现的对比

3.4.1 基于时间的比较

线性表的操作主要有查找、插入、删除三类操作。

对于查找操作有基于序号的查找，即存取线性表中 i 号数据元素。由于数组有随机存取的特性，在线性表的顺序存储实现中可以在 $\Theta(1)$的时间内完成；而在链式存储中由于需要从头结点开始顺着链表才能取得，无法在常数时间内完成，因此顺序存储优于链式存储。查找操作还有基于元素的查找，即线性表是否包含某个元素、元素的序号是多少，这类操作线性表的顺序存储与链式存储都需要从线性表中序号为 0 的元素开始依次查找，因此两种实现的性能相同。综上所述，如果在线性表的使用中主要操作是查找，那么应当选用顺序存储实现的线性表。

对于基于数据元素的插入、删除操作而言，当使用数组实现相应操作时，首先需要采用顺序查找定位相应数据元素，然后才能插入、删除，并且在插入、删除过程又要移动大量元素；相对而言链表的实现只需要在定位数据元素的基础上，简单的修改几个指针即可完成，因此链式存储优于顺序存储。对于基于序号的插入、删除操作，因为在顺序存储中平均需要移动一半元素；而在链式存储中不能直接定位，平均需要比较一半元素才能定位。因此顺序存储与链式存储性能相当。综上所述，如果在线性表的使用中主要操作是插入、删除操作，那么选用链式存储的线性表为佳。

3.4.2 基于空间的比较

线性表的顺序存储，其存储空间是预先静态分配的，虽然在实现的过程中可以动态扩展数组空间，但是如果线性表的长度变化范围较大，空间在使用过程中由于会存在大量空闲空间，使得存储空间的利用率不高。而线性表的链式存储，其结点空间是动态分配的，不会存在存储空间没有完全利用的情况。因此当线性表长度变化较大时，宜采用链式存储结构。

当线性表的数据元素结构简单，并且线性表的长度变化不大时。由于链式存储结构使用了额外的存储空间来表示数据元素之间的逻辑关系，因此针对数据域而言，指针域所占比重较大；而在线性表的顺序存储结构中，没有使用额外的存储空间来表示数据元素之间的逻辑关系，尽管有一定的空闲空间没有利用，但总体而言由于线性表长度变化不大，因此没有利用的空间所占比例较小。所以当线性表数据元素结构简单，长度变化不大时可以考虑采用顺序存储结构。

3.5 链接表

3.5.1 基于结点的操作

在 3.1.2 小节给出的线性表抽象数据类型中，其提供的操作主要是指对线性表中的数据元素及其序号的。例如插入操作就是基于序号和元素进行的，insert(i, e)是在序号为 i 的地方插入元素，insertBefore 、与 insertAfter 是在某个数据元素之前或之后插入新的元素。这种基于序号的操作实际上并不适合采用（单向或双向）链表来实现，因为为了在链表中定位数据元素或序号，我们不得不沿着结点间的 next（或 pre）引用，从链表前端（双向链表也可以从后端）开始逐一扫描。

我们考察一种经常需要完成的操作：顺序的将线性表中每个数据元素都访问一遍。如果使用链式存储实现的线性表ListSLinked所提供的get(i)操作来实现，则需要$O(n^2)$时间。因为在使用链表实现取i号数据元素的操作时，需要将结点的引用从链表前端向后移动i次，而取 i+1 号数据元素时不能在上一次操作——取i号数据元素——的过程中受益，而必须重新从链表前端开始定位，则访问线性表中每个元素一次所需要的总时间为 $0+1+2+\ldots+n-1=O(n^2)$。这一时间复杂度是难以接受的。

实际上，除了通过序号来访问线性结构中的元素，还可通过其他途径来得到线性结构中的元素。例如我们能够直接通过结点来访问数据元素，通过 3.3.1 中定义的结点接口，我们看到结点实际上可以看成是可以存取数据元素的容器，数据元素与存放它的容器是一一对应的。如果能够取得结点的引用，则可以取得相应结点存储的数据元素，并且在实际应用中的许多情况下更希望以结点作为参数来完成某些操作。

如果能够以结点作为参数，那么就可以在$O(1)$时间内定位结点的地址，进而可以在更短的时间内完成相应的操作。例如如果能够直接定位在链表中进行插入和删除结点的前驱，那么相应的插入和删除操作都可以在$O(1)$完成。

3.5.2 链接表接口

链接表可以看成是一组结点序列以及基于结点进行操作的线性结构的抽象，或者说是对链表的抽象。

在链接表中提供基于结点的操作时，有一个问题需要考虑：需要将多少链接表的实现细节暴露给使用它的程序员？如果将单链表或双向链表的细节，例如结点结构、首结点引用或尾结点引用都提供给程序员。这样做可以使得程序员可以直接访问数据并修改内部链表结构（例如通过 next 和 pre 引用），但是基于安全性和面向对象的封装原则，我们并不这样做。那么如何在向用户提供相关链表结点引用的基础上，却可以保证用户不会通过该引用对链表的内部结构直接进行访问或修改呢？这实际上可以通过 3.1.1 定义的 Node 接口来实现，因为任何链表结点（单链表结点、双向链表结点都实现了 Node 接口）都可被 Node 类型的变量引用，而 Node 接口中只有存取数据元素的方法，因此程序员即可以存取数据，又不能对内部链表结构进行修改。

代码 3-10 给出链接表支持的操作接口定义。

代码 3-10 链接表接口

```
public interface LinkedList {
    //查询链接表当前的规模
```

```
    public int getSize();
    //判断列表是否为空
    public boolean isEmpty();
    //返回第一个结点
    public Node first() throws OutOfBoundaryException;
    //返回最后一结点
    public Node last() throws OutOfBoundaryException;
    //返回 p 之后的结点
    public Node getNext(Node p) throws InvalidNodeException, OutOfBoundaryException;
    //返回 p 之前的结点
    public Node getPre(Node p) throws InvalidNodeException, OutOfBoundaryException;
    //将 e 作为第一个元素插入链接表,并返回 e 所在结点
    public Node insertFirst(Object e);
    //将 e 作为最后一个元素插入列表,并返回 e 所在结点
    public Node insertLast(Object e);
    //将 e 插入至 p 之后的位置,并返回 e 所在结点
    public Node insertAfter(Node p, Object e) throws InvalidNodeException;
    //将 e 插入至 p 之前的位置,并返回 e 所在结点
    public Node insertBefore(Node p, Object e) throws InvalidNodeException;
    //删除给定位置处的元素，并返回之
    public Object remove(Node p) throws InvalidNodeException;
    //删除首元素，并返回之
    public Object removeFirst() throws OutOfBoundaryException;
    //删除末元素，并返回之
    public Object removeLast() throws OutOfBoundaryException;
    //将处于给定位置的元素替换为新元素，并返回被替换的元素
    public Object replace(Node p, Object e) throws InvalidNodeException;
    //元素迭代器
    public Iterator elements();
}
```

其中最后一个方法 elements()在 3.6 中介绍，InvalidNodeException 是当作为参数的结点不合法时抛出的异常。定义如下：

代码 3-11 InvalidNodeException 异常

```
public class InvalidNodeException extends RuntimeException {
    public InvalidNodeException(String err) {
        super(err);
    }
}
```

结点 p 在以下情况下可以认为是不合法的：

- p==null；
- p 在链接表中不存在；
- 在调用方法 getPre(p)时，p 已经是第一个存有数据的结点；
- 在调用方法 getNext(p)时，p 已经是最后一个存有数据的结点。

3.5.3 基于双向链表实现的链接表

在 3.3.2 小节中，为了实现双向链表结构，曾经在代码 3-8 中定义了双向链表结点结构 DLNode。由于 DLNode 实现了 Node 接口，所以 DLNode 本身就是一个结点，对内部的链表而言就是组成链表的一部份，而对于外部而言就是可以存取数据元素的容器。

在使用双向链表实现链接表时，为使编程更加简洁，我们使用带两个哑元结点的双向链表来实现链接表。其中一个是头结点，另一个是尾结点，它们都不存放数据元素，头结点的 pre 为空，而尾结点的 Next 为空。如此构成的双向链表结构如图 3-15 所示。

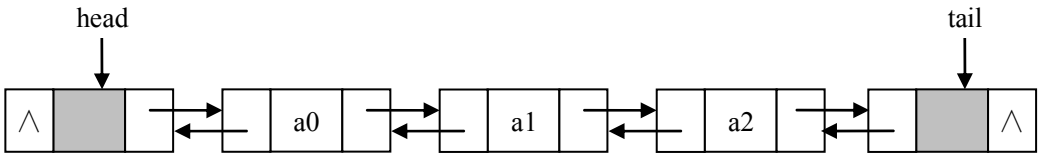


图 3-15 带头尾结点的双向链表

在具有头尾结点的双向链表中插入和删除结点，无论插入和删除的结点位置在何处，因为首尾结点的存在，插入、删除操作都可以被归结为 3.3.2 小节中介绍的在双向链表某个中间结点的插入和删除；并且因为首尾结点的存在，整个链表永远不会为空，因此在插入和删除结点之后，也不用考虑链表由空变为非空或由非空变为空的情况下 head 和 tail 的指向问题；从而简化了程序。

代码 3-12 基于双向链表实现的链接表

```
public class LinkedListDLNode implements LinkedList {
    private int size;       //规模
    private DLNode head;//头结点,哑元结点
    private DLNode tail;//尾结点,哑元结点
    public LinkedListDLNode() {
        size = 0;
        head = new DLNode();//构建只有头尾结点的链表
        tail = new DLNode();
        head.setNext(tail);
        tail.setPre(head);
    }

    //辅助方法，判断结点 p 是否合法，如合法转换为 DLNode
    protected DLNode checkPosition(Node p) throws InvalidNodeException {
        if (p==null)
            throw new InvalidNodeException("错误：p 为空。");
        if (p==head)
            throw new InvalidNodeException("错误：p 指向头节点，非法。");
        if (p==tail)
            throw new InvalidNodeException("错误：p 指向尾结点，非法。");
        DLNode node = (DLNode)p;
        return node;
    }
```

```
//查询链接表当前的规模
public int getSize() {
	return size;
}

//判断链接表是否为空
public boolean isEmpty() {
	return size==0;
}

//返回第一个结点
public Node first() throws OutOfBoundaryException{
	if (isEmpty())
		throw new OutOfBoundaryException("错误：链接表为空。");
	return head.getNext();
}

//返回最后一结点
public Node last() throws OutOfBoundaryException{
	if (isEmpty())
		throw new OutOfBoundaryException("错误：链接表为空。");
	return tail.getPre();
}

//返回 p 之后的结点
public Node getNext(Node p)throws InvalidNodeException,OutOfBoundaryException {
	DLNode node = checkPosition(p);
	node = node.getNext();
	if (node==tail)
		throw new OutOfBoundaryException("错误：已经是链接表尾端。");
	return node;
}

//返回 p 之前的结点
public Node getPre(Node p) throws InvalidNodeException, OutOfBoundaryException {
	DLNode node = checkPosition(p);
	node = node.getPre();
	if (node==head)
		throw new OutOfBoundaryException("错误：已经是链接表前端。");
	return node;
}

//将 e 作为第一个元素插入链接表
public Node insertFirst(Object e) {
```

```
        DLNode node = new DLNode(e,head,head.getNext());
        head.getNext().setPre(node);
        head.setNext(node);
        size++;
        return node;
    }

    //将 e 作为最后一个元素插入列表,并返回 e 所在结点
    public Node insertLast(Object e) {
        DLNode node = new DLNode(e,tail.getPre(),tail);
        tail.getPre().setNext(node);
        tail.setPre(node);
        size++;
        return node;
    }

    //将 e 插入至 p 之后的位置,并返回 e 所在结点
    public Node insertAfter(Node p, Object e) throws InvalidNodeException {
        DLNode node = checkPosition(p);
        DLNode newNode = new DLNode(e,node,node.getNext());
        node.getNext().setPre(newNode);
        node.setNext(newNode);
        size++;
        return newNode;
    }

    //将 e 插入至 p 之前的位置,并返回 e 所在结点
    public Node insertBefore(Node p, Object e) throws InvalidNodeException {
        DLNode node = checkPosition(p);
        DLNode newNode = new DLNode(e,node.getPre(),node);
        node.getPre().setNext(newNode);
        node.setPre(newNode);
        size++;
        return newNode;
    }

    //删除给定位置处的元素，并返回之
    public Object remove(Node p) throws InvalidNodeException {
        DLNode node = checkPosition(p);
        Object obj = node.getData();
        node.getPre().setNext(node.getNext());
        node.getNext().setPre(node.getPre());
        size--;
        return obj;
```

```
    }

    //删除首元素，并返回之
    public Object removeFirst() throws OutOfBoundaryException{
        return remove(head.getNext());
    }

    //删除末元素，并返回之
    public Object removeLast() throws OutOfBoundaryException{
        return remove(tail.getPre());
    }

    //将处于给定位置的元素替换为新元素，并返回被替换的元素
    public Object replace(Node p, Object e) throws InvalidNodeException {
        DLNode node = checkPosition(p);
        Object obj = node.getData();
        node.setData(e);
        return obj;
    }

    //元素迭代器
    public Iterator elements() {
        return new LinkedListIterator(this);
    }
}
```

代码 3-12 说明：LinkedListDLNode 中共有 3 个成员变量，其中 head 和 tail 分别指向双向链表中空的头结点和尾结点，它们本身并不存储数据元素；size 用来标明当前链接表中数据元素的个数，使用该成员变量可以在 O(1)时间内返回链接表的规模，而不用从头至尾计数元素的个数。除此之外，LinkedListDLNode 中其他各个方法的正确性不难理解，并且各个方法的时间复杂度均为 O(1)。通过上述代码可以看到在使用结点作为参数时，链表实现插入、删除操作的优越性就明显的体现出来。

3.6 迭代器

迭代器（Iterator）是程序设计的一种模式，它属于设计模式中的行为模式，它的功能是提供一种方法顺序访问一个聚集对象中各个元素，而又不需暴露该对象的内部表示。

多个对象聚在一起形成的总体称之为聚集（Aggregate），聚集对象是能够包容一组对象的容器对象。聚集依赖于聚集结构的抽象化，具有复杂性和多样性。例如数组就是一种最基本的聚集。

聚集对象需要提供一种方法，允许用户按照一定的顺序访问其中的所有元素。而迭代器提供了一个访问聚集对象中各个元素的统一接口，简单的说迭代器就是对遍历操作的抽象。在一个迭代器中一般需要提供以下操作：

表 3-1 迭代器支持的操作

序号	方法	功能描述
(1)	first()	输入参数：无 返回参数：无 功能：将游标指向到第一个元素
(2)	next()	输入参数：无 返回参数：无 功能：将游标指向下一个元素。
(3)	isDone()	输入参数：无 返回参数：boolean 功能：判断迭代器中是否还有剩余的元素。
(4)	currentItem()	输入参数：无 返回参数：Object 对象 功能：返回迭代器当前数据元素。

根据以上定义的操作，我们先给出迭代器的 Java 接口。

代码 3-13 迭代器接口

```
public interface Iterator {
    //移动到第一个元素
    public void first();
    //移动到下一个元素
    public void next();
    //检查迭代器中是否还有剩余的元素
    public boolean isDone();
    //返回当前元素
    public Object currentItem();
}
```

迭代器的实现可以根据不同的聚集对象给出不同的实现，下面我们结合聚集对象 LinkedList 对象，来实现针对 LinkedList 的迭代器。代码 3-14 给出了完整的实现代码。

代码 3-14 LinkedListIterator，基于 LinkedList 聚集对象的迭代器实现

```
public class LinkedListIterator implements Iterator {
    private LinkedList list;//链接表
    private Node current;//当前结点
    //构造方法
    public LinkedListIterator(LinkedList list) {
        this.list = list;
        if (list.isEmpty())          //若列表为空
            current = null;          //则当前元素置空
        else
            current = list.first();//否则从第一个元素开始
    }

    //移动到第一个元素
    public void first(){
        if (list.isEmpty())          //若列表为空
```

```
            current = null;        //则当前元素置空
        else
            current = list.first();//否则从第一个元素开始
    }

    //移动到下一个元素
    public void next() throws OutOfBoundaryException{
        if (isDone())
            throw new OutOfBoundaryException("错误：已经没有元素。");
        if (current==list.last()) current = null;   //当前元素后面没有更多元素
        else current = list.getNext(current);
    }

    //检查迭代器中是否还有剩余的元素
    public boolean isDone() { return current==null; }

    //返回当前元素
    public Object currentItem() throws OutOfBoundaryException{
        if (isDone())
            throw new OutOfBoundaryException("错误：已经没有元素。");
        return current.getData();
    }
}
```

代码 3-14 说明：由于本迭代器是基于链接表聚集对象的，因此在类中有一个成员变量为链接表对象引用；除此之外还有一个用于返回当前元素的结点对象引用。LinkedListIterator 代码中各方法的正确性不难理解，且各个方法均在 O(1)时间内完成。

在有了基于链接表聚集对象的迭代器实现以后，就可以对链接表中的数据使用迭代器接口提供的方法进行完整的遍历了。例如在代码 3-12 基于双向链表实现的链接表代码中可以对外提供一个访问所有数据元素的迭代器，即代码 3-12 中最后一个 elements()方法实现的功能。

第四章　栈与队列

栈和队列是两种重要的数据结构。从栈与队列的逻辑结构上来说，它们也是线性结构，与线性表不同的是它们所支持的基本操作是受到限制的，它们是操作受限的线性表，是一种限定性的数据结构。

4.1 栈

4.1.1 栈的定义及抽象数据类型

栈（stack）又称堆栈，它是运算受限的线性表，其限制是仅允许在表的一端进行插入和删除操作，不允许在其他任何位置进行插入、查找、删除等操作。表中进行插入、删除操作的一端称为**栈顶（top）**，栈顶保存的元素称为**栈顶元素**。相对的，表的另一端称为**栈底（bottom）**。

当栈中没有数据元素时称为空栈；向一个栈插入元素又称为**进栈**或**入栈**；从一个栈中删除元素又称为**出栈**或**退栈**。由于栈的插入和删除操作仅在栈顶进行，后进栈的元素必定先出栈，所以又把堆栈称为**后进先出表**（Last In First Out，简称 LIFO）。图 4-1 显示了一个堆栈及数据元素插入和删除的过程。

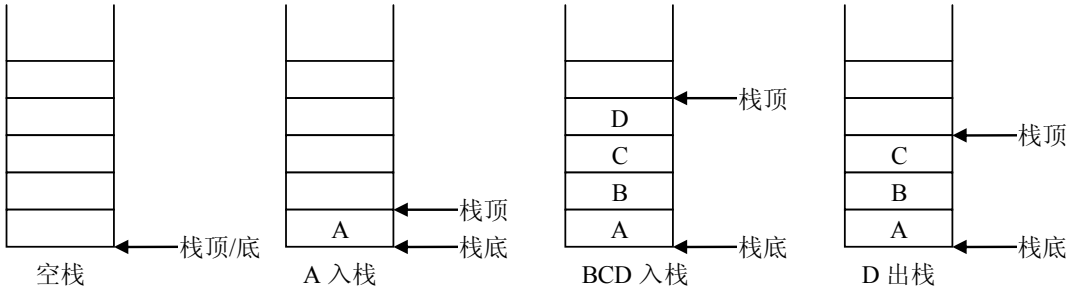


图 4-1 堆栈及入栈和出栈

在图 4-1 中当 ABCD 均已入栈之后，出栈时得到的序列为 DCBA，这就是“后进先出”。在解决实际问题时，如果碰到了数据的使用具有“后进先出”的特性，就预示着可以使用堆栈来存储和使用这些数据。

堆栈的基本操作除了进栈、出栈操作外，还有判空、取栈顶元素等操作。下面给出堆栈的抽象数据类型定义。

ADT Stack {

数据对象： $D = \{a_i \mid a_i \in D0,\ i=0, 1, 2 \ldots n-1,\ D_0$为某一数据对象$\}$

数据关系： $R = \{<a_i, a_{i+1}> \mid a_i, a_{i+1} \in D,\ i=0, 1, 2 \ldots n-2\}$

基本操作：

序号	方法	功能描述
(1)	getSzie ()	输入参数：无 返回参数：非负整数 功能：返回堆栈的大小，即数据元素的个数。

(2)	isEmpty ()	输入参数：无 返回参数：boolean 功能：如果堆栈为空返回 true，否则返回 false。
(3)	push(e)	输入参数：Object 对象 e 返回参数：无 功能：数据元素 e 入栈。
(4)	pop()	输入参数：无 返回参数：Object 对象 功能：栈顶元素出栈。
(5)	peek()	输入参数：无 返回参数：Object 对象 功能：获取栈顶元素，但不出栈。

} ADT Stack

对应于堆栈的抽象数据类型，代码 4-1 给出了完整的 Java 接口。

代码 4-1 Stack 接口

```
public interface Stack {
    //返回堆栈的大小
    public int getSize();
    //判断堆栈是否为空
    public boolean isEmpty();
    //数据元素 e 入栈
    public void push(Object e);
    //栈顶元素出栈
    public Object pop() throws StackEmptyException;
    //取栈顶元素
    public Object peek() throws StackEmptyException;
}
```

其中涉及的异常类定义如下：

代码 4-2 StackEmptyException 堆栈为空时出栈或取栈顶元素抛出此异常

```
public class StackEmptyException extends RuntimeException{
    public StackEmptyException(String err) {
        super(err);
    }
}
```

4.1.2 栈的顺序存储实现

和线性表类似，堆栈也有两种基本的存储结构：顺序存储结构和链式存储结构。

顺序栈是使用顺序存储结构实现的堆栈，即利用一组地址连续的存储单元依次存放堆栈中的数据元素。由于堆栈是一种特殊的线性表，因此在线性表的顺序存储结构的基础上，选择线性表的一端作为栈顶即可。根据数组操作的特性，选择数组下标大的一端，即线性表顺序存储的表尾来作为栈顶，此时入栈、出栈等操作可以在O(1)时间完成。

由于堆栈的操作都在栈顶完成，因此在顺序栈的实现中需要附设一个指针 top 来动态的

指示栈顶元素在数组中的位置。通常 top 可以用栈顶元素所在数组下标来表示，top= -1 时表示空栈。图 4-1 就可以看成是一个顺序栈。

堆栈在使用过程中所需的最大空间很难估计，因此，一般来说在构造堆栈时不应设定堆栈的最大容量。一种合理的做法和线性表的实现类似，先为堆栈分配一个基本容量，然后在实际的使用过程中，当堆栈的空间不够时再倍增存储空间，这个过程所需的时间均摊到每个数据元素时间为 $\Theta(1)$，不会影响操作实现的时间复杂度。

代码 4-3 给出了基于以上思想实现的堆栈。

代码 4-3 Stack 的顺序存储实现

```
public class StackArray implements Stack {
    private final int LEN = 8;      //数组的默认大小
    private Object[] elements;      //数据元素数组
    private int top;                //栈顶指针

    public StackArray() {
        top = -1;
        elements = new Object[LEN];
    }

    //返回堆栈的大小
    public int getSize() {
        return top+1;
    }

    //判断堆栈是否为空
    public boolean isEmpty() {
        return top<0;
    }

    //数据元素 e 入栈
    public void push(Object e) {
        if (getSize()>=elements.length) expandSpace();
        elements[++top] = e;
    }
    private void expandSpace(){
        Object[] a = new Object[elements.length*2];
        for (int i=0; i<elements.length; i++)
            a[i] = elements[i];
        elements = a;
    }

    //栈顶元素出栈
    public Object pop() throws StackEmptyException {
        if (getSize()<1)
            throw new StackEmptyException("错误，堆栈为空。");
```

```
        Object obj = elements[top];
        elements[top--] = null;
        return obj;
    }

    //取栈顶元素
    public Object peek() throws StackEmptyException {
        if (getSize()<1)
            throw new StackEmptyException("错误，堆栈为空。");
        return elements[top];
    }
}
```

代码 4-3 说明：以上基于数组实现堆栈代码的正确性不难理解。由于有 top 指针的存在，所以 getSize、isEmpty 均可在 O(1)时间内完成；push、pop、peek 除掉用 getSize 外都执行常数基本操作，因此它们的运行时间也是 O(1)。

4.1.3 栈的链式存储实现

链栈即采用链表作为存储结构实现的栈。当采用单链表存储线性表后，根据单链表的操作特性选择单链表的头部作为栈顶，此时，入栈、出栈等操作可以在 O(1)内完成。由于堆栈的操作只在线性表的一端进行，在这里使用带头结点的单链表或不带头结点的单链表都可以。使用带头结点的单链表时，结点的插入和删除都在头结点之后进行；使用不带头结点的单链表时，结点的插入和删除都在链表的首结点上进行。

下面以不带头结点的单链表为例实现堆栈，读者可以对照完成使用带头结点的单链表的实现。图 4-2 给出了使用不带头结点的单链表实现堆栈的示意图。

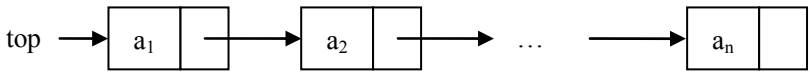


图 4-2 链栈示意图

图 4-2 中，top 为栈顶结点引用，始终指向当前栈顶元素所在结点。若 top 为 Null，则表示空栈。入栈操作是在 top 所指结点之前插入新的结点，对照图 4-2 可以看到，当链表为空和不为空时，入栈操作的实现都一样，可以使用以下语句来实现

```
SLNode q = new SLNode(e,top);  //结点 q 的 next 域指向 top，不管 top 是否为 Null
top = q;
```

同样对于出栈操作而言，不管堆栈在删除栈顶元素之后，栈是否为空，出栈操作都是将 top 后移。

代码 4-4 给出了堆栈的链式存储实现。

代码 4-4 Stack 的链式存储实现

```
public class StackSLinked implements Stack {
    private SLNode top;      //链表首结点引用
    private int size;            //栈的大小
    public StackSLinked() {
        top = null;      size = 0;
    }
```

```
    //返回堆栈的大小
    public int getSize() {
        return size;
    }

    //判断堆栈是否为空
    public boolean isEmpty() {
        return size==0;
    }

    //数据元素 e 入栈
    public void push(Object e) {
        SLNode q = new SLNode(e,top);
        top = q;
        size++;
    }

    //栈顶元素出栈
    public Object pop() throws StackEmptyException {
        if (size<1)
            throw new StackEmptyException("错误，堆栈为空。");
        Object obj = top.getData();
        top = top.getNext();
        size--;
        return obj;
    }

    //取栈顶元素
    public Object peek() throws StackEmptyException {
        if (size<1)
            throw new StackEmptyException("错误，堆栈为空。");
        return top.getData();
    }
}
```

与代码 4-3 类似，代码 4-4 的正确性不难理解。并且在代码 4-4 中，所有的操作都是在 O(1)时间内完成。

4.2 队列

4.2.1 队列的定义及抽象数据类型

队列（queue）简称队，它同堆栈一样，也是一种运算受限的线性表，其限制是仅允许在表的一端进行插入，而在表的另一端进行删除。在队列中把插入数据元素的一端称为**队尾**

（**rear**），删除数据元素的一端称为**队首（front）**。向队尾插入元素称为**进队或入队**，新元素入队后成为新的队尾元素；从队列中删除元素称为**离队或出队**，元素出队后，其后续元素成为新的队首元素。

由于队列的插入和删除操作分别在队尾和队首进行，每个元素必然按照进入的次序离队，也就是说先进队的元素必然先离队，所以称队列为**先进先出表**（First In First Out,简称FIFO）。队列结构与日常生活中排队等候服务的模型是一致的，最早进入队列的人，最早得到服务并从队首离开；最后到来的人只能排在队列的最后，最后得到服务并最后离开。

下面给出队列的抽象数据类型定义。

ADT Queue{

数据对象：D = {a_i | a_i∈D0，i=0, 1, 2…n-1，D_0为某一数据对象}

数据关系：R = {<a_i, a_{i+1}> | a_i, a_{i+1}∈D，i=0, 1, 2 … n-2}

基本操作：

序号	方法	功能描述
(1)	getSzie ()	输入参数：无 返回参数：非负整数 功能：返回堆栈的大小，即数据元素的个数。
(2)	isEmpty ()	输入参数：无 返回参数：boolean 功能：如果堆栈为空返回 true，否则返回 false。
(3)	enqueue (e)	输入参数：Object 对象 e 返回参数：无 功能：数据元素 e 入队。
(4)	dequeue ()	输入参数：无 返回参数：Object 对象 功能：栈顶元素出队。
(5)	peek()	输入参数：无 返回参数：Object 对象 功能：获取队首元素，但不出队。

} ADT Queue

对应于队列的抽象数据类型，代码 4-5 给出了完整的 Java 接口。

代码 4-5 Queue 接口

```
public interface Queue {
    //返回队列的大小
    public int getSize();
    //判断队列是否为空
    public boolean isEmpty();
    //数据元素 e 入队
    public void enqueue(Object e);
    //队首元素出队
    public Object dequeue() throws QueueEmptyException;
    //取队首元素
    public Object peek() throws QueueEmptyException;
}
```

其中涉及的异常类定义如下：

代码 4-6 QueueEmptyException 队列为空时出队或取队首元素抛出此异常

```
public class QueueEmptyException extends RuntimeException {

    public QueueEmptyException(String err) {
        super(err);
    }
}
```

4.2.2 队列的顺序存储实现

在队列的顺序存储实现中，我们可以将队列当作一般的表用数组加以实现，但这样做的效果并不好。尽管我们可以用一个指针 last 来指示队尾，使得 enqueue 运算可在 O(1)时间内完成，但是在执行 dequeue 时，为了删除队首元素，必须将数组中其他所有元素都向前移动一个位置。这样，当队列中有 n 个元素时，执行 dequeue 就需要 O(n)时间。

为了提高运算的效率，我们用另一种方法来表达数组中各单元的位置关系。设想数组 A[0.. capacity-1]中的单元不是排成一行，而是围成一个圆环，即 A[0]接在 A[capacity-1]的后面。这种意义下的数组称为循环数组，如图 4-3 所示。

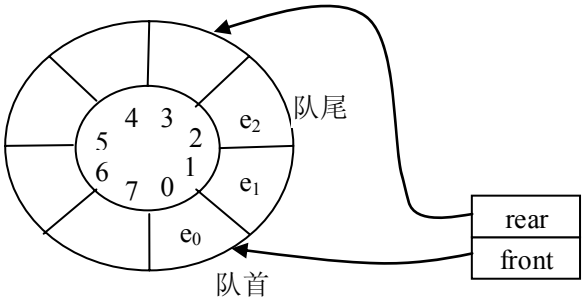


图 4-3 循环数组

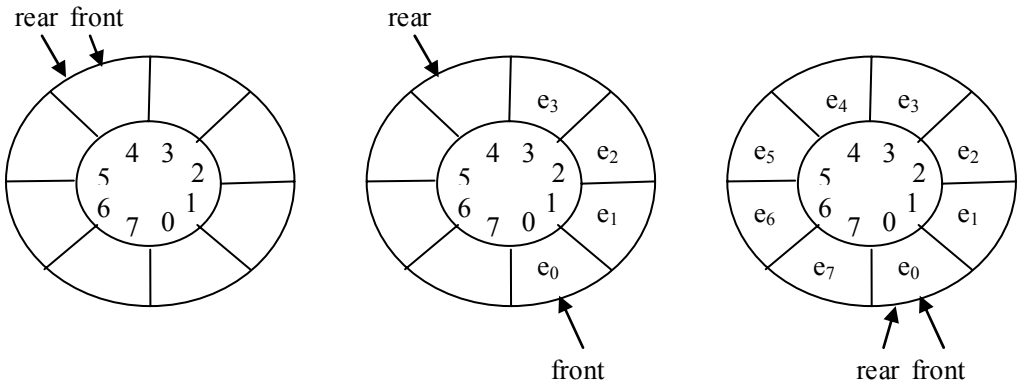
用循环数组实现的队列称为循环队列，我们将循环队列中从队首到队尾的元素按逆时针方向存放在循环数组中一段连续的单元中。并且直接用队首指针 front 指向队首元素所在的单元，用队尾指针 rear 指向队尾元素所在单元的后一个单元。如图 4-3 所示，队首元素存储在数组下标为 0 的位置，front=0；队尾元素存储在数组下标为 2 的位置，rear=3。

当需要将新元素入队时，可在队尾指针指示的单元中存入新元素，并将队尾指针 rear 按逆时针方向移一位。出队操作也很简单，只要将队首指针 front 依逆时针方向移一位即可。容易看出，用循环数组来实现队列可以在 O(1)时间内完成 enqueue 和 dequeue 运算。执行一系列的入队与出队运算，将使整个队列在循环数组中按逆时针方向移动。

当然队首和队尾指针也可以有不同的指向，例如也可以用队首指针 front 指向队首元素所在单元的前一个单元，或者用队尾指针 rear 指向队尾元素所在单元的方法来表示队列在循环数组中的位置。但是不论使用哪一种方法来指示队首与队尾元素，我们都要解决一个细节问题，即如何表示满队列和空队列。

下面以图 4-3 所示的表示方法来说明这个问题。在图 4-3 中用队首指针front指向队首元素所在的单元，用队尾指针rear指向队尾元素所在单元的后一个单元。如此在图 4-4（b）中所示循环队列中，队首元素为e_0，队尾元素为e_3。当e_4、e_5、e_6、e_7相继进入队列后，如图 4-4（c）所示，队列空间被占满，此时队尾指针追上队首指针，有rear = front。反之，如果从图 4-4（b）所示的状态开始，e_0、e_1、e_2、e_3相继出队，则得到空队列，如图 4-4（a）所示，此时队首指针追上队尾指针，所以也有front = rear。可见仅凭front与rear是否相等无法判断队

列的状态是“空”还是“满”。解决这个问题可以有两种处理方法：一种方法是少使用一个存储空间，当队尾指针的下一个单元就是队首指针所指单元时，则停止入队。这样队尾指针就不会追上队首指针，所以在队列满时就不会有front = rear。这样一来，队列满的条件就变为(rear+1)% capacity = front，而队列判空的条件不变，仍然为front = rear。另外一种解决这个问题的方法是增设一个标志，以区别队列是“空”还是“满”，例如增设size变量表明队列中数据元素的个数，如果size = Max则队列满。



（a）　　（b）　　（c）

图 4-4 循环队列

表 4-1 总结了上述分析的结果。

表 4-1 循环队列中各关键量

	不使用 size 标记队列元素个数	使用 size 标记队列元素个数
队首元素	elements[front]	elements[front]
队尾元素	elements[(rear-1) % capacity]	elements[(rear-1) % capacity]
队空	rear=front	size=0
队满	(rear+1)%capacity=front	size=capacity

注：其中 elements 为存放队列元素的数组。

下面以少使用一个存储单元的方案实现循环队列。

代码 4-7 Queue 的顺序存储实现

```
public class QueueArray implements Queue {
    private static final int CAP = 7;//队列默认大小
    private Object[] elements;     //数据元素数组
    private int capacity;          //数组的大小 elements.length
    private int front;             //队首指针,指向队首
    private int rear;              //队尾指针,指向队尾后一个位置
    public QueueArray() {
        this(CAP);
    }
    public QueueArray(int cap){
        capacity = cap + 1;
        elements = new Object[capacity];
        front = rear = 0;
    }

    //返回队列的大小
```

```
    public int getSize() {
        return (rear -front+ capacity)%capacity;
    }

    //判断队列是否为空
    public boolean isEmpty() {
        return front==rear;
    }

    //数据元素 e 入队
    public void enqueue(Object e) {
        if (getSize()==capacity-1) expandSpace();
        elements[rear] = e;
        rear = (rear+1)%capacity;
    }
    private void expandSpace(){
        Object[] a = new Object[elements.length*2];
        int i = front;    int j = 0;
        while (i!=rear){   //将从 front 开始到 rear 前一个存储单元的元素复制到新数组
            a[j++] = elements[i];
            i = (i+1)%capacity;
        }
        elements = a;
        capacity = elements.length;
        front = 0;        rear = j;  //设置新的队首、队尾指针
    }

    //队首元素出队
    public Object dequeue() throws QueueEmptyException {
        if (isEmpty())
            throw new QueueEmptyException("错误：队列为空");
        Object obj = elements[front];
        elements[front] = null;
        front = (front+1)%capacity;
        return obj;
    }

    //取队首元素
    public Object peek() throws QueueEmptyException {
        if (isEmpty())
            throw new QueueEmptyException("错误：队列为空");
        return elements[front];
    }
}
```

代码 4-7 说明：在 QueueArray 类中成员变量 CAP 是用来以默认大小生成队列，由于我们采用损失一个存储单元来区分队列空与满的两种不同状态，因此实际的数组大小要比队列最大容量大 1。为了代码的简洁，在 QueueArray 类中引入成员变量 capacity 表示数组的大小，即 capacity = elements.length。除此之外各操作的实现不难理解，并且每个操作的实现方法其时间复杂度 T(n)= O (1)。

4.2.3 队列的链式存储实现

队列的链式存储可以使用单链表来实现。为了操作实现方便，这里采用带头结点的单链表结构。根据单链表的特点，选择链表的头部作为队首，链表的尾部作为队尾。除了链表头结点需要通过一个引用来指向之外，还需要一个对链表尾结点的引用，以方便队列的入队操作的实现。为此一共设置两个指针，一个队首指针和一个队尾指针，如图 4-5 所示。队首指针指向队首元素的前一个结点，即始终指向链表空的头结点，队尾指针指向队列当前队尾元素所在的结点。当队列为空时，队首指针与队尾指针均指向空的头结点。

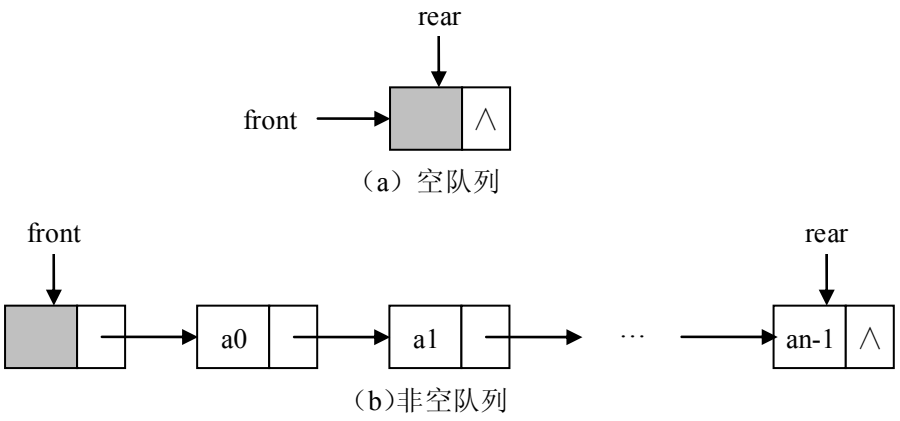


图 4-5 队列的链式存储结构

代码 4-8 给出了队列链式存储的操作实现。

代码 4-8 Queue 的链式存储实现

```
public class QueueSLinked implements Queue {
    private SLNode front;
    private SLNode rear;
    private int size;
    public QueueSLinked() {
        front = new SLNode();
        rear = front;
        size = 0;
    }

    //返回队列的大小
    public int getSize() {
        return size;
    }

    //判断队列是否为空
```

```
    public boolean isEmpty() {
        return size==0;
    }

    //数据元素 e 入队
    public void enqueue(Object e) {
        SLNode p = new SLNode(e,null);
        rear.setNext(p);
        rear = p;
        size++;
    }

    //队首元素出队
    public Object dequeue() throws QueueEmptyException {
        if (size<1)
            throw new QueueEmptyException("错误：队列为空");
        SLNode p = front.getNext();
        front.setNext(p.getNext());
        size--;
        if (size<1) rear = front;  //如果队列为空,rear 指向头结点
        return p.getData();
    }

    //取队首元素
    public Object peek() throws QueueEmptyException {
        if (size<1)
            throw new QueueEmptyException("错误：队列为空");
        return front.getNext().getData();
    }
}
```

代码 4-8 的正确性不难理解，并且所有操作的实现算法，其时间复杂度 T(n)=O(1)。

4.3 堆栈的应用

堆栈所具有的后进先出特性，使得堆栈成为程序设计中非常有用的工具。本节将讨论堆栈应用的典型例子。

4.3.1 进制转换

进制转换是一种常见的数值计算问题，例如将十进制数转换成八进制数。实现进制转换的一种简单方法是重复以下两步，直到 N 等于 0。

```
X = N mod d        //其中 mod 为求余运算
N = N div d        //其中 div 为整除运算
```

最后得到的一系列余数就是转换后的结果。

例如：$(2007)_{10} = (3727)_8$，其运算过程如下：

8	2007	余数
8	250	7
8	31	2
8	3	7
	0	3

可以看到上述过程是从低位到高位产生 8 进制数的各个数位，而在输出时，一般来说都是从高位到低位进行输出，这正好产生数位的顺序相反。换一个说法就是，越晚生成的数位越早需要输出，结果数位的使用具有后出现先使用的特点，因此生成的结果数位可以使用一个堆栈来存储，然后从栈顶开始依次输出即可得到相应的转换结果。

算法 4-1 实现了十进制数到八进制数的转换。

算法 4-1 baseConversion

输入：十进制正整数 i

输出：打印相应八进制数

代码：

```
public void baseConversion(int i){
    Stack s = new StackSLinked();
    while (i>0){
        s.push(i%8+"");
        i = i/8;
    }
    while (!s.isEmpty()) System.out.print((String)s.pop());
}
```

4.3.2 括号匹配检测

假设表达式中包含三种括号：圆括号、方括号和花括号，并且它们可以任意相互嵌套。例如{[{}]([])}或[{()[]}]等为正确格式，而{[( ])}或({[()})等均为不正确的格式。

该问题可按“期待匹配消解”的思想来设计算法，对表达式中的每一个左括号都期待一个相应的右括号与之匹配，表达式中越迟出现并且没有得到匹配的左括号期待匹配的程度越高。不是期待出现的右括号则是非法的。它具有天然的后进先出的特点。

于是可以如下设计算法：算法需要一个堆栈，在读入字符的过程中，如果是左括号，则直接入栈，等待相匹配的同类右括号；若读入的是右括号，且与当前栈顶左括号匹配，则将栈顶左括号出栈，如果不匹配则属于不合法的情况。另外如果碰到一个右括号，而堆栈为空，说明没有左括号与之匹配，属于非法情况；或者字符读完，而堆栈不为空，说明有左括号没有得到匹配，也属于非法情况。当字符读完同时堆栈为空，并且在匹配过程中没有发现不匹配的情况，说明所有的括号是匹配的。

算法 4-2 bracketMatch

输入：字符串 str

输出：boolean，匹配结果

代码：

```
public boolean bracketMatch(String str) {
    Stack s = new StackSLinked();
    for (int i=0;i<str.length();i++)
    {
        char c = str.charAt(i);
        switch (c)
        {
            case '{':
            case '[':
            case '(': s.push(Integer.valueOf(c)); break;
            case '}':
                if (!s.isEmpty()&& ((Integer)s.pop()).intValue()=='{')
                    break;
                else return false;
            case ']':
                if (!s.isEmpty()&& ((Integer)s.pop()).intValue()=='[')
                    break;
                else return false;
            case ')':
                if (!s.isEmpty()&& ((Integer)s.pop()).intValue()=='(')
                    break;
                else return false;
        }
    }
    if (s.isEmpty()) return true;
    else  return false;
}
```

4.3.3 迷宫求解

求解从迷宫中的起点到某个终点的路径是一个有趣的问题，如图 4-6 所示。使用计算机求解迷宫问题时，通常采用的方法是系统的尝试所有可能的路径：即从起点出发，顺着某个方向向前探索，例如向当前位置的左边探索，若当前位置除向左之外还有其他方向的没有被访问过的邻接点，则在向左探索之前，按固定的次序记录下当前位置其他可能的探索方向；若当前位置向左不能再走下去，则换到当前位置的其他方向进行探索；如果当前位置所有方向的探索均结束，却没有到达终点，则沿路返回当前位置的前一个位置，并在此位置还没有探索过的方向继续进行探索；直到所有可能的路径都被探索到为止。

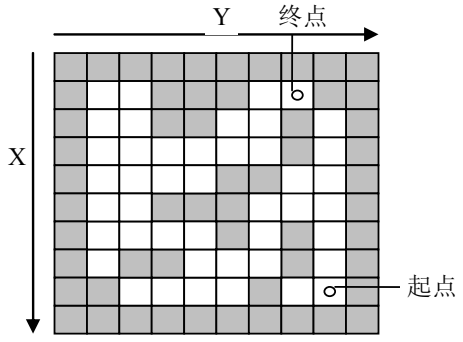


图 4-6 迷宫

为了保证在任何位置上都能原路返回，因此需要使用一个后进先出的存储结构来保存从起点到当前位置的路径以及在路径上各位置还可能进行探索的方向。因此在迷宫问题中使用堆栈是自然的。

首先在计算机中可以使用一个二维字符数组来表示图 4-6 所示的迷宫。我们使用字符'1'来表示迷宫中的墙体，即灰色的方块；用字符'0'来表示迷宫中可以通过的道路，即白色的方块。按上述方法，图 4-6 所示的迷宫可以用图 4-7（a）的二维字符数组表示。

1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	1	0	0	1	1
1	0	0	1	1	0	0	1	0	1
1	0	0	0	0	0	0	1	0	1
1	0	0	0	0	1	1	0	0	1
1	0	0	1	1	1	0	0	0	1
1	0	0	0	0	1	0	1	0	1
1	0	1	1	0	0	0	1	0	1
1	1	0	0	0	0	1	0	0	1
1	1	1	1	1	1	1	1	1	1

（a）

1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	1	*	*	1	1
1	0	0	1	1	*	*	1	0	1
1	*	*	*	*	*	0	1	0	1
1	*	0	0	0	1	1	0	0	1
1	*	0	1	1	1	*	*	*	1
1	*	*	*	*	1	*	1	*	1
1	0	1	1	*	*	*	1	*	1
1	1	0	0	0	0	1	0	*	1
1	1	1	1	1	1	1	1	1	1

（b）

图 4-7 迷宫的二维数组模拟

其次，求解迷宫的算法思想可以描述为：

初始化，将起点加入堆栈；
while(堆栈不空){
　　取出栈顶位置作为当前位置；
　　如果 当前位置是终点，
　　则　　使用堆栈记录的路径标记从起点至终点的路径；
　　否则{　　按照向下、右、上、左的顺序将当前位置下一个可以探索的位置入栈；
　　　　//从堆栈取出的探索方向顺序则是左、上、右、下
　　　　如果 当前位置没四周均不可通
　　　　则　　当前位置出栈；
　　}
}

迷宫中当前位置的下一个可以探索的点是**未曾走到过**的位置及其有待探索的下一个位置，即要求该位置不但是通道块，而且不在当前路径上，也不是曾经纳入到路径中或有待探索的通道块。这可以通过对每一个位置设置一个标志，来表明该位置是否可以作为下一个可探索的位置。

为了在算法中可以对每一个位置进行操作，下面先定义迷宫中的每一个位置。

代码 4-8 迷宫单元的定义

```
private class Cell{
    int x = 0; //单元所在行
    int y = 0; //单元所在列
    boolean visited = false;  //是否访问过
    char c = ' ';                  //是墙('1')、可通路('0')或起点到终点的路径('*')
    public Cell(int x, int y, char c, boolean visited){
        this.x = x;       this.y = y;
        this.c = c;this.visited = visited;
```

```
    }
}
```

算法 4-3 实现了迷宫中从起点到终点路径的求解。

算法 4-3 mazeExit

输入：表示迷宫的字符数组，起点、终点的坐标。并且字符数组及坐标有效。

输出：找到从起点到终点的路径

代码：

```
    public void mazeExit(char[][] maze,int sx,int sy, int ex,int ey){
        Cell[][] cells = createMaze(maze); //创建化迷宫
        printMaze(cells);                  //打印迷宫
        Stack s = new StackSLinked();      //构造堆栈
        Cell startCell = cells[sx][sy];    //起点
        Cell endCell = cells[ex][ey];      //终点
        s.push(startCell);                 //起点入栈
        startCell.visited = true;          //标记起点已被访问
        while (!s.isEmpty()){
            Cell current = (Cell)s.peek();
            if (current==endCell){   //路径找到
                while(!s.isEmpty()){
                    Cell cell = (Cell)s.pop();//沿路返回将路径上的单元设为*
                    cell.c = '*';
                    //堆栈中与 cell 相邻的单元才是路径的组成部分，除此之外，
                    //堆栈中还有记录下来但是未继续向下探索的单元，
                    //这些单元直接出栈
                    while(!s.isEmpty()&&!isAdjoinCell((Cell)s.peek(),cell)) s.pop();
                }
                System.out.println("找到从起点到终点的路径。");
                printMaze(cells);
                return;
            } else {   //如果当前位置不是终点
                int x = current.x;
                int y = current.y;
                int count = 0;
                if(isValidWayCell(cells[x+1][y])){//向下
                    s.push(cells[x+1][y]); cells[x+1][y].visited = true; count++;}
                if(isValidWayCell(cells[x][y+1])){//向右
                    s.push(cells[x][y+1]); cells[x][y+1].visited = true; count++;}
                if(isValidWayCell(cells[x-1][y])){ //向上
                    s.push(cells[x-1][y]); cells[x-1][y].visited = true; count++;}
                if(isValidWayCell(cells[x][y-1])){ //向左
                    s.push(cells[x][y-1]); cells[x][y-1].visited = true; count++;}
                if (count==0) s.pop();//如果是死点，出栈
            }//end of if
        }//end of while
```

```
        System.out.println("没有从起点到终点的路径。");
    }

    private void printMaze(Cell[][] cells){
        for (int x=0;x<cells.length;x++){
            for (int y=0;y<cells[x].length;y++)
                System.out.print(cells[x][y].c);
            System.out.println();
        }
    }
    private boolean isAdjoinCell(Cell cell1, Cell cell2){
        if (cell1.x==cell2.x&&Math.abs(cell1.y-cell2.y)<2) return true;
        if (cell1.y==cell2.y&&Math.abs(cell1.x-cell2.x)<2) return true;
        return false;
    }
    private boolean isValidWayCell(Cell cell){
        return cell.c=='0'&&!cell.visited;
    }
    private Cell[][] createMaze(char[][] maze){
        Cell[][] cells = new Cell[maze.length][];
        for (int x=0;x<maze.length;x++){
            char[] row = maze[x];
            cells[x] = new Cell[row.length];
            for (int y=0; y<row.length;y++)
                cells[x][y] = new Cell(x,y,maze[x][y],false);
        }
        return cells;
    }
```

在算法 4-3 中还需要注意的一点是：因为迷宫四周有墙，因此从当前位置向四周探索时，数组下标不会越界；如果迷宫四周没有墙，则在向四周探索时，要验证探索位置的下标是否越界。如果以图 4-7（a）中的二维字符数组以及(8, 8)为起点坐标、以(1, 7)为终点坐标作为输入，算法 4-3 的输出如图 4-7（b）所示。

第五章　递归

递归是在计算机科学、数学等领域运用非常广泛的一种方法。使用递归的方法解决问题，一般具有这样的特征：我们在寻求一个复杂问题的解时，不能立即给出答案，然而从一个规模较小的相同问题的答案开始，却可以较为容易的求解复杂的问题。

本章介绍两种基本的基于递归的算法设计技术，即基于归纳的递归和分治法。

5.1 递归与堆栈

5.1.1 递归的概念

递归（recursion）是指在定义自身的同时又出现了对自身的引用。如果一个算法直接或间接地调用自己，则称这个算法是一个递归算法。

任何一个有意义的递归算法总是由两部分组成：递归调用与递归终止条件。下面我们来看一个递归算法的简单例子。

例 5-1　计算一个整数 n 的阶乘。通过阶乘的数学定义我们知道：

$$n! = \begin{cases} 1 & n=0 \\ n\,(n-1)! & n>0 \end{cases}$$

实现它的递归算法如下：

算法 5-1 factorial

输入：正整数 n

输出：n!

代码：

```
public int factorial (int n) {                    // 1.
    if (n == 0)                                   // 2.
        return 1;                                 // 3.
    else                                          // 4.
        return n* factorial(n-1);                 // 5.
}                                                 // 6.
```

在算法 5-1 中第 2 行是判断是否满足递归终止条件，如果满足则执行第 3 行，否则进行递归调用执行第 5 行。在这里可以看到递归调用与递归终止条件在递归算法中缺一不可，如果没有递归终止条件那么递归将会无休止的进行下去；而没有递归调用，则递归算法就不成其为递归算法。因此在编写递归算法时一定要注意这两个方面的内容。下面我们再看一个递归算法的例子。

例 5-2　计算以 x 为底的 n 次幂，其中 n 为非负整数。计算整数次幂可以简单的使用一个循环迭代 n 次，每让 x 乘以自身即可。但是这样算法的时间复杂度为 Θ(n)，效率较低。下面我们设计一个新的算法，可以使得时间复杂度为 Θ(log n)。

因为x^n可以写成如下形式：

$$x^n = \begin{cases} 1 & n=0 \\ \left(x^{\lfloor n/2 \rfloor}\right)^2 & n>0,\ n \text{ 为偶数} \\ \left(x^{\lfloor n/2 \rfloor}\right)^2 \cdot x & n>0,\ n \text{ 为奇数} \end{cases}$$

这显然是一个递归定义，其中当 n 为 0 时是递归终止条件；否则如果 n 大于 0，则需要进行递归调用，只不过在进行递归调用时需要分两种不同情况分别进行处理。算法 5-2 实现了这一过程。

算法 5-2 power

输入：整数 x、非负整数 n

输出：x^n

代码：

```
public int power (int x, int n) {          // 1.
    int y;                                 // 2.
    if (n == 0)                            // 3.  递归终止条件
        y = 1;                             // 4.
    else  {                                // 5.
        y = power (x, n/2);                // 6.  递归调用
        y = y * y;                         // 7.
        if (n%2 == 1)  y = y * x;          // 8.
    }                                      // 9.
    return y;                              //10.
 }                                         // 11.
```

如果将乘法作为基本操作，则算法 5-2 的时间复杂度函数可以如下表示：

$$\begin{cases} T(1) = 1 \\ T(n) = T(n/2) + 1 \quad n>1 \end{cases}$$

这是一个非常简单的递推关系的求解问题，因为每进行一次调用，n 变为原来的一半，因此总共的递归调用次数为 Θ(log n)，因此

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= (T(n/4) + 1) + 1 \\ &= ((T(n/8) + 1)+1) +1 \\ &= (\ldots(T(1) + 1) + \ldots +1) + 1 \\ &= \Theta(\log n) \end{aligned}$$

通过例 5-1 和例 5-2 可以看到，递归算法的结构清晰明了、易于阅读，并且算法的正确性可以很容易的使用数学归纳法得到证明。这些都为算法设计和程序调试带来了很大方便，它是算法设计中一种非常有用的技术。在实际应用中使用递归可以解决以下多方面的问题：⑴ 问题本身的定义就是递归的，例如许多数学定义就是递归的。⑵ 问题本身虽然不是递归定义的，但是它所用到的数据结构是递归的，例如链表、树就可以看成是递归定义的数据结构。⑶ 问题的解法满足递归的性质，例如在本章后面将要介绍的一些问题。

5.1.2 递归的实现与堆栈

在第四章中介绍的堆栈还有一个非常重要的应用，即在程序设计语言中实现递归。我们知道在递归算法中会递归调用自身，因此在递归算法的执行过程中会多次进行自我调用。那么这个调用过程是如何实现的呢？

为了说明自身的递归调用，我们先看任意两个函数（不同程序设计语言对“函数”称谓不同，在这里我们不妨都称之为函数）之间进行调用的情形。

通常在一个函数执行过程中需要调用另一个函数时，在运行被调用函数之前系统通常需要完成如下工作：⑴ 对调用函数的运行现场进行保护，主要是参数与返回地址等信息的保存；⑵ 创建被调用函数的运行环境；⑶ 将程序控制转移到被调用函数的入口。在被调用函数执行结束之后，返回调用函数之前，系统同样需要完成 3 件工作：⑴ 保存被调函数的返回结果；⑵ 释放被调用函数的数据区；⑶ 依照保存的调用函数的返回地址将程序控制转移到调用函数。

如果上述函数调用的过程中发生了新的调用，即被调函数在执行完成之前又调用了其他函数，此时构成了多个函数的嵌套调用。当发生嵌套调用时按照后调用先返回的原则处理，如此则形成了一个保存函数运行时环境变量的“后进先出”的使用过程，因此整个函数调用期间的相关信息的保存需要使用一个堆栈来实现。系统将整个程序运行时需要的数据空间安排在一个堆栈中，每当调用一个函数时就为它在栈顶分配一个存储区，每当从一个函数返回时就释放它的存储区。

一个递归算法的实现实际上就是多个相同函数的嵌套调用。

下面我们用算法 5-1：factorial 来说明递归的实现过程。假设 n=3，那么递归调用的过程如图 5-1 所示。

```
public int factorial (int n) {          // 1.
    if (n == 0)                         // 2.
        return 1;                       // 3.
    else                                // 4.
        return n* factorial(n-1);       // 5.
}                                       // 6.
```

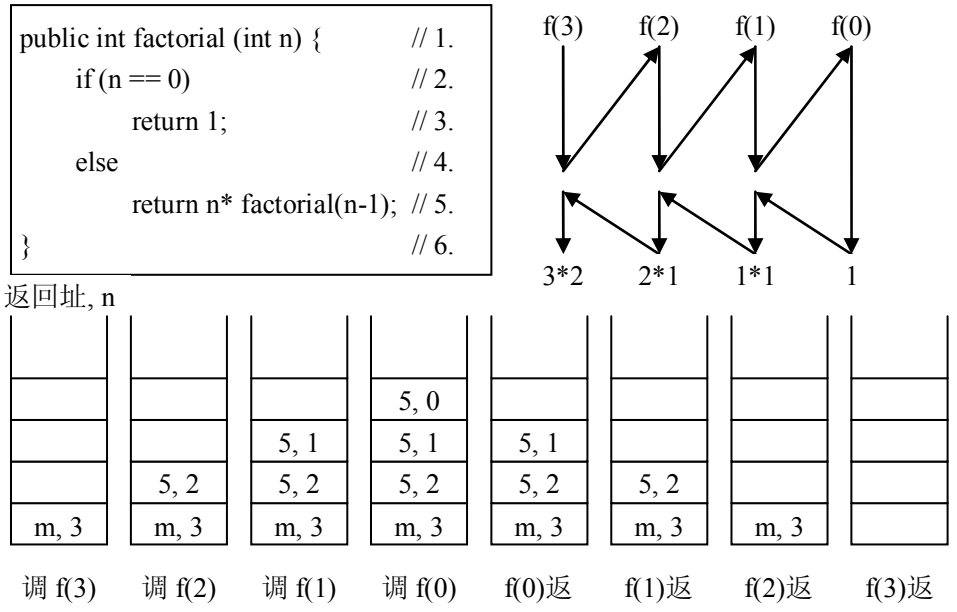


图 5-1 递归调用示例

通过上面的内容，我们介绍了递归算法的实现原理。但是同时我们看到递归方法在某些情况下却并不一定是最高效的方法，主要原因在于递归方法过于频繁的函数调用和参数传递，这会使系统有较大的开销。在某些情况下，若采用循环或递归算法的非递归实现，将会大大提高算法的实际执行效率。当然这也并不意味着不建议使用递归方法解决问题，递归仍然是非常有用和广泛使用的技术。

5.2 基于归纳的递归

基于归纳的递归是一种较为简单并且也是一种基本的递归算法设计方法。它的主要思想是把数学归纳法应用于算法设计之中。

对于一个规模为 n 的问题 P(n)，归纳法的步骤是：

Ⅰ. 基本项：A_1是问题P(1)的解

Ⅱ. 递归项：对于所有的k，1<k<n ，若A_k是问题P(k)的解，则存在f使得f(A_k)是P(k+1)的解。其中f是对A_k的某种运算或处理。

因此，为求问题P(n)的解A_n，可以先求P(n-1)的解A_{n-1}然后对A_{n-1}进行f运算或处理。而为求P(n-1)的解，先求P(n-2)得解，如此不断的递归求解，直到P(1)为止。

例 5-3 求解 n 阶汉诺塔问题。汉诺塔问题是由法国数学家 Edouard Lucas 在 1883 年发明的。n 阶汉诺塔问题可以描述为：假设有 X、Y、Z 三个塔座，初始时有 n 个大小不一的盘子按照从小到大的次序放在塔座 X 上。现在要求将塔座 X 上的所有盘子移动到塔座 Z 上并保持原来的顺序，在移动过程中要满足以下要求：在塔座之间一次只能移动一个盘子并且任何时候大盘子都不能放到小盘子上。

基本项：若只有一个盘子，则不需要使用过渡塔座，直接把它放到目的塔座即可。

递归项：如果多于一个盘子，则需要将塔座 X 上的 1 到 n-1 个盘子使用 Z 作为过渡塔座放到塔座 Y 上，然后将第 n 个盘子（最后一个盘子）放到塔座 Z，最后将塔座 Y 上的 n-1 个盘子使用塔座 X 作为过渡放到塔座 Z。

算法 5-3 hanoi

输入：正整数 n

输出：n 阶 Hanoi 塔的移动步骤

代码：

```
public void hanio (int n, char x, char y, char z){
    if (n==1) move ( x, n, z);
    else {
        hanio (n-1, x, z, y);
        move (x, n, z);
        hanio(n-1, y, x, z);
    }
}

private void move(char x, int n, char y) {
    System.out.println ("Move " + n + " from " + x + " to " + y);
}
```

下面来分析算法 hanoi 的时间复杂度。这里以盘子的移动作为基本运算，通过对上面算法的分析，得到算法的时间复杂度函数可以写成以下形式：

$$\begin{cases} T(1) = 1 \\ T(n) = 2T(n-1) + 1 \quad n>1 \end{cases}$$

hanoi算法在n取 1 到 5 时的运行时间为：1、3、7、15、31，我们发现T(n)满足如下关系：

$T(n) = 2^n - 1$。对此我们使用数学归纳法进行证明：

当n=1 时，$T(1) = 2^n - 1 = 1$，满足时间复杂度函数。

假设当n<=k时$T(n) = 2^n - 1$ 成立，则$T(k) = 2^k - 1$。

当n=k+1 时，$T(n) = T(k+1) = 2T(k) + 1 = 2(2^k - 1) + 1 = 2^{k+1} - 1 = 2^n - 1$ 成立，因此：当n>=1 时$T(n) = 2^n - 1$。

最后我们得到算法hanoi的时间复杂度$T(n) = O(2^n)$。

下面我们来看另外一个例子。

例 5-4　编写一个算法输出 n 个布尔量的所有可能组合。

每个布尔量有真和假两种取值，分别对应 1，0。对于n个布尔量有 2^n种组合，每一种均为n位。

基本项：如果 n 为 1，则只需要输出 0 和 1 即可。

递归项：n个布尔量的 2^n种所有不同的组合可以看成是 $2*2^{n-1}$种组合，其中 2^{n-1}种组合是 n-1 个布尔量的所有组合，每种组合包含n-1 位。这样n个布尔变量的所有组合是在n-1 个布尔变量的每种组合的基础上加上 1 或 0 而分别得到的结果。

在具体的实现中使用一个数组 b 来存放 n 位组合的每一个分量。

算法 5-4 coding

输入：正整数 n

输出：n 位布尔量的所有组合

代码：（数组下标从 0 开始，因此调用方法时参数中的 n 应当取数组 b 的下标上限。）

```
public void coding (int[] b, int n) {
    if (n==0) {
        b[n] = 0;outBn(b);
        b[n] = 1;outBn(b);
    }
    else {
        b[n] = 0; coding(b,n-1);
        b[n] = 1; coding(b,n-1);
    }
}

private void outBn (int[] b) {
    for (int i=0;i<b.length;i++)
        System.out.print(b[i]);
    System.out.println();
}
```

下面来分析算法 coding 的时间复杂度。假设不考虑 n 位布尔量的输出，以赋值作为基本运算，通过对上面算法的分析，得到算法的时间复杂度函数可以写成以下形式：

$$\begin{cases} T(1) = 2 \\ T(n) = 2T(n-1) + 2 \quad n>1 \end{cases}$$

同样可以使用数学归纳法证明$T(n) = 2^{n+1}-2$。因此算法coding的时间复杂度$T(n) =O(2^n)$。

5.3 递推关系求解

5.3.1 求解递推关系的常用方法

■ 数学归纳法

从上面介绍的算法可以看出，递归算法的时间复杂度是使用递推关系给出的。求解递推关系的一种方法如同例 5-3、例 5-4 那样可以先观察前几项，猜测 T(n)的通项，然后用数学归纳法证明，最后得出时间复杂度。使用这种方法的一个问题是要有足够的观察力猜出通项，这使得使用这种方法存在一定难度。

■ 迭代法

求解递推关系的另外一种方法就是迭代法，用这个方法估计递归方程解的渐近阶不要求推测解的渐近表达式，但要求较多的代数运算。方法的思想是迭代地展开递归方程的右端，使之成为一个非递归的和式，然后通过对和式的估计来达到对方程左端即方程的解的估计。

例 5-5 求解递推关系 T(n) = 3T(n/4) + n，T(0) = 1。

$T(n) = 3T(n/4) + n$

$= n + 3(n/4 + 3T(n/4^2))$

$= n + 3(n/4 + 3(n/4^2 + 3T(n/4^3)))$

$= n + 3(n/4 + 3(n/4^2 + 3(n/4^3 + \ldots + 3(n/4^i + 3T(0))\ldots)))$

$= n + \frac{3}{4}n + \frac{3^2}{4^2}n + \ldots + \frac{3^i}{4^i}n + 3^{i+1}T(0)$

$< \sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i n + 3^{1+\log_4 n}T(0)$

$= 4n + 3n^{\log_4 3}$

$= O(n)$

其中$i = \log_4 n$，并且由于$T(n) > n = \Omega(n)$，因此$T(n) = \Theta(n)$。

从这个例子可以看到迭代法导致繁杂的代数运算。

■ 递归树

在一棵递归树中，每个节点代表了一组递推表达式中函数符号所表示的子问题的代价。我们求出树中每层节点的代价之和就得到一层的代价和，然后我们将树中每层的代价和求出来就可以确定所有递归调用的代价。它对描述分治算法的递归方程特别有效。

下面我们用一个例子来说明使用递归树求解递推关系这种方法。

例 5-6 求解递推关系$T(n) = 3T(n/4) + cn^2$，c为大于 0 的常数，T(1)= 1。

使用递归树求解该递推关系的过程如图 5-2。

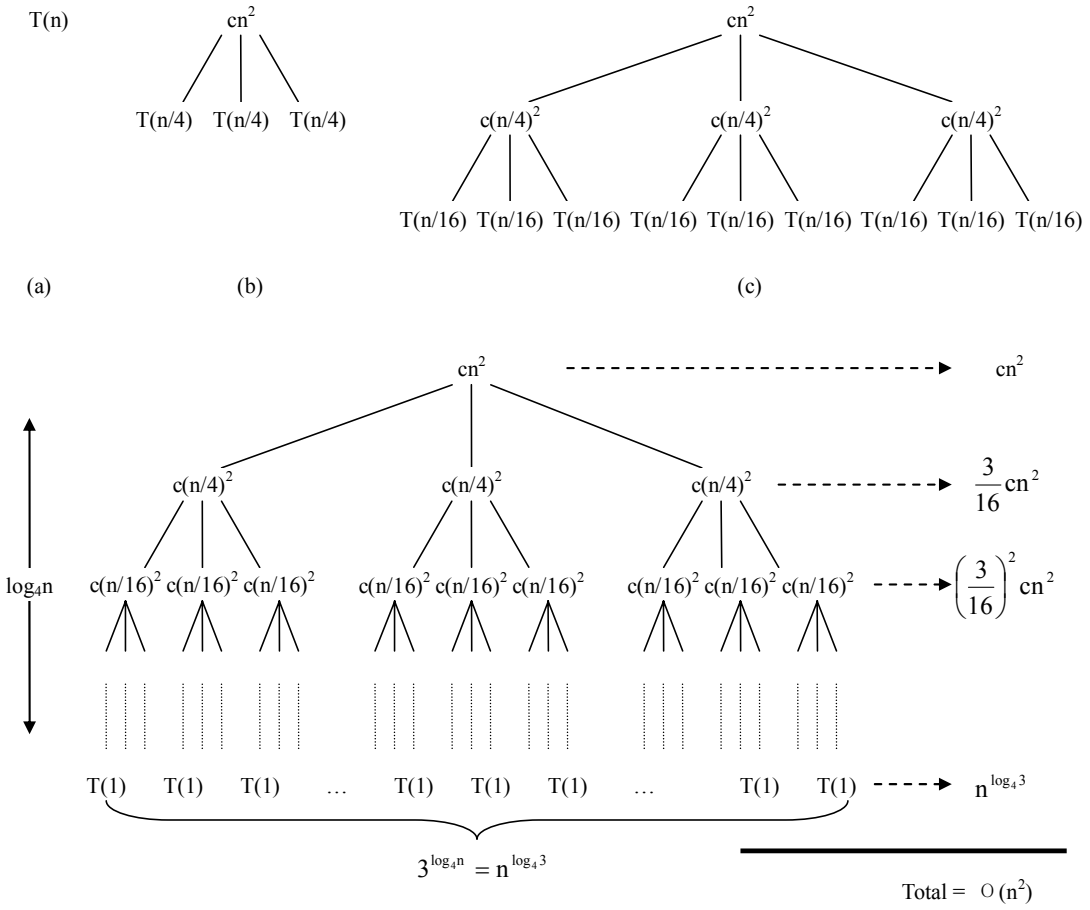


图 5-2 递归树示例

通过图 5-1 我们看到：

首先每向下进行一次递归调用，问题的规模会变为原来的 1/4，最后问题的规模会变为 1，在这个过程中一共向下进行了多少层的递归调用呢？不妨设为i，则$n/4^i = 1$，即$i = \log_4 n$ 。

其次，由于每个节点代表的问题规模是上一层节点的 1/4，不妨假设上层节点的问题规模为k，那么本层每个节点的问题规模为k/4。在节点向下进行递归调用时，上层节点的代价为ck^2，而本层每个节点的代价为$c(k/4)^2$，即上层节点的 1/16。由于本层节点数是上层的 3 倍，所以本层所有节点代价和是上层节点代价和的 3/16。

最后每向下一层进行递归调用，每层节点数是上层的 3 倍，而一共进行了$\log_4 n$次，因此最下一层的节点数是$1 * 3^{\log_4 n} = n^{\log_4 3}$。

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + T(1)\cdot n^{\log_4 3}$$

$$< \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3}$$

$$= \frac{16}{13}cn^2 + n^{\log_4 3}$$

$$=O(n^2)$$

并且由于$T(n) > cn^2 = \Omega(n^2)$，所以$T(n) = \Theta(n^2)$

除了上面介绍的基本方法，下面向读者介绍求解一些特定形式的递推关系的方法，这些递推形式是在本书中会经常出现的。如果读者需要了解更多有关递推关系求解方法的内容，可以参考组合数学中的相关内容。

5.3.2 线性齐次递推式的求解

我们把注意力主要放在一阶和二阶的线性齐次递推关系式的求解上。

一阶线性齐次递推关系式的解可以直接得到，令$f(n) = a\,f(n-1)$，假定序列从$f(0)$开始，容易得出$f(n) = a^n f(0)$。

假设二阶线性齐次递推关系式的形式为$f(n) = a_1 f(n-1) + a_2 f(n-2)$，假定序列从$f(0)$以及$f(1)$开始，那么对于该递推关系的求解方法为：

⑴ 求解特征方程$x^2 - a_1 x - a_2 = 0$,令其两个根为r_1和r_2

⑵ 按如下公式求出 $f(n)$

$$f(n) = \begin{cases} c_1 r_1^n + c_2 r_2^n & r_1 \neq r_2 \\ c_1 r^n + c_2 n r^n & r_1 = r_2 = r \end{cases}$$

⑶ 将$f(0)$和$f(1)$代入第二步求得的结果，计算出c_1和c_2

例 5-7 求解递推关系 $f(n) = f(n-1) + 2\,f(n-2)$ 且 $f(0) = 1$，$f(1) = 2$。

特征方程是$x^2 - x - 2 = 0$，有$r_1=-1$，$r_2=2$，所以递推解为$f(n) = c_1(-1)^n + c_2(2)^n$。为求出$c_1$和$c_2$解以下两个方程：

$f(0) = c_1 + c_2 = 1$，$f(1) = -c_1 + 2c_2 = 2$

得到：$c_1 = 0$ 和$c_2 = 1$，所以$f(n) = 2^n$。

例 5-8 考虑 Fibonacci 序列 1, 1, 2, 3, 5, 8, …它可以使使用递推关系表示为 $f(n) = f(n-1) + f(n-2)$ 且 $f(1) = 1$，$f(2) = 1$。为简化讨论可引入 $f(0) = 0$。

特征方程是$x^2 - x - 1 = 0$，有$r_1=(1+\sqrt{5})/2$，$r_2=(1-\sqrt{5})/2$，所以递推解为

$$f(n) = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

为求出c_1和c_2解以下两个方程：

$$f(0) = c_1 + c_2 = 0,\quad f(1) = c_1\left(\frac{1+\sqrt{5}}{2}\right) + c_2\left(\frac{1-\sqrt{5}}{2}\right) = 1$$

得到：$c_1 = 1/\sqrt{5}$ 和$c_2 = -1/\sqrt{5}$，所以

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

当 n 足够大时，第二项趋近于 0，因此当 n 足够大时

$$f(n) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

5.3.3 非齐次递推关系的解

同样，在这里我们主要关注在算法分析中常用的两种非齐次递推关系上。

第一种也是最简单的非齐次递推关系是：

$f(n) = f(n-1) + g(n)$，$n>0$

容易得出递推式的解是：$f(n) = f(0) + \sum_{i=1}^{n} g(i)$

下面可虑另一个递推关系

$f(n) = g(n)\, f(n-1) + h(n)$，$n>0$

为了解这个关系定义一个新函数 $f'(n)$，令

$$f(n) = \begin{cases} g(n)g(n-1)\cdots g(1)f'(n), & n > 0 \\ f'(0) & n = 0 \end{cases}$$

将 $f(n)$ 与 $f(n-1)$代入原递推关系式，得到

$g(n)\, g(n-1)\ldots g(1)\, f'(n) = g(n)\, (g(n-1)\ldots g(1)\, f'(n-1)) + h(n)$

上式简化为

$f'(n) = f'(n-1) + h(n)/\, g(n)\, g(n-1)\ldots g(1)$

因此

$$f'(n) = f'(0) + \sum_{i=1}^{n} \frac{h(i)}{g(i)\, g(i-1)\ldots g(1)}$$

最后求出

$$f(n) = g(n)\, g(n-1)\ldots g(1)\left(f(0) + \sum_{i=1}^{n} \frac{h(i)}{g(i)\, g(i-1)\ldots g(1)}\right)$$

例 5-9　考虑例 5-4 中的递推关系 $T(n) = 2T(n-1) + 2$，$T(1) = 2$。

为方便求解可以引入$T(0) = 0$，令$T(n) = 2^n\, T'(n)$，$T(0)= T'(0)= 0$。那么

$2^n\, T'(n) = 2\, (2^{n-1}\, T'(n-1)) + 2$

化简上式为

$T'(n) = T'(n-1) + 2/2^n$

它的解是

$$T'(n) = T'(0) + \sum_{i=1}^{n} \frac{2}{2^i} = \sum_{i=1}^{n} \frac{2}{2^i}$$

最后解出

$$T(n) = 2^n T'(n) = 2^n \sum_{i=1}^{n} \frac{2}{2^i} = 2^{n+1}-2$$

这与在例 5-4 中得到的结果是一致的。

例 5-10 求解递推关系 $f(n) = n f(n-1) + n!$，$f(0) = 0$。

令 $f(n) = n! f'(n)$，$f(0) = f'(0)= 0$。那么

$$n! f'(n) = n ((n-1)! f'(n-1)) + n!$$

化简上式为

$$f'(n) = f'(n-1) + 1$$

它的解是

$$f'(n) = f'(0) + n = n$$

最后解出

$$f(n) = n! f'(n) = nn!$$

5.3.4 Master Method

Master Method 为求解如下形式的递推式提供了简单的方法。

$$T(n) = aT(n/b) + f(n) \qquad (5.1)$$

其中a、b为常数，并且$a \geq 1$, $b > 1$；$f(n)$是正的确定函数。

在 Master Method 中分 3 种不同的情况分别给出问题的解。(5.1)是一类分治法的时间复杂性所满足的递归关系，即一个规模为 n 的问题被分成规模均为 n/b 的 a 个子问题，递归地求解这 a 个子问题，然后通过对这 a 个子问题的解的综合，得到原问题的解。如果用 T(n)表示规模为 n 的原问题的复杂性，用 $f(n)$表示把原问题分成 a 个子问题和将 a 个子问题的解综合为原问题的解所需要的时间，我们便有递推关系式(5.1)。关于分治法的具体内容我们将在 5.4 中作详细介绍。

Master Method 依赖于下面的定理 5.1

定理 5.1 设 $a \geq 1$ 和 $b>1$ 是常数，$f(n)$是定义在非负整数上的一个确定的非负函数。又设 T(n)也是定义在非负整数上的一个非负函数，且满足递推关系式(5.1)。递推关系式(5.1)中的 n/b 可以是$\lfloor n/b \rfloor$，也可以是$\lceil n/b \rceil$。那么我们有如下的 T(n)渐近估计式：

1. 对于某常数 $\varepsilon>0$，如果 $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$，那么 $T(n) = \Theta\left(n^{\log_b a}\right)$
2. 如果 $f(n) = \Theta\left(n^{\log_b a}\right)$，那么 $T(n) = \Theta\left(n^{\log_b a} \log n\right)$
3. 对于某常数 $\varepsilon>0$，如果 $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$，且对于某常数 $c<1$ 和所有充分大的正整数 n 有 $a f(n/b) \leq c f(n)$，那么 $T(n) = \Theta(f(n))$。

证明：略

通过对定理 5.1 的分析，读者可能已经注意到，这里涉及的三类情况，都是拿 $f(n)$与 $n^{\log_b a}$ 作比较。定理直观地告诉我们，递归关系式解的渐近阶由这两个函数中的较大者决定。在第一类情况下，函数 $n^{\log_b a}$ 较大，则 $T(n) = \Theta\left(n^{\log_b a}\right)$；在第三类情况下，函数 $f(n)$较大，则

$T(n) = \Theta(f(n))$；在第二类情况下，两个函数一样大，则乘以 n 的对数作为因子，此时 $T(n) = \Theta\left(n^{\log_b a} \log n\right)$。

在这个定理中需要特别注意的是在第一类情况下 $f(n)$不仅必须比 $n^{\log_b a}$ 小，而且必须是多项式地比 $n^{\log_b a}$ 小；在第三类情况下 $f(n)$不仅必须比 $n^{\log_b a}$ 大，而且必须是多项式地比 $n^{\log_b a}$ 大，还要满足附加的条件：对于某常数 $c<1$ 和所有充分大的正整数 n，有 $a f(n/b) \le c f(n)$。这个附加条件的直观含义是 a 个子问题的再分解和再综合所需要的时间最多与原问题的分解和综合所需要的时间同阶。我们在一般情况下将碰到的以多项式为界的函数基本上都满足这个条件。

例 5-11 求解递推关系 $T(n) = 4T(n/2) + n$

$a = 4$，$b = 2 \Rightarrow n^{\log_b a} = n^2$

$f(n) = n$

Case 1：$f(n) = O\left(n^{2-\varepsilon}\right)$，其中 $\varepsilon=1>0$

$\therefore T(n) = \Theta(n^2)$

例 5-12 求解递推关系 $T(n) = T(n/2) + 1$

$a = 1$，$b = 2 \Rightarrow n^{\log_b a} = n^0 = 1$

$f(n) = 1$

Case 2：$f(n) = \Theta(1)$

$\therefore T(n) = \Theta(\log n)$

例 5-13 求解递推关系 $T(n) = 2T(n/4) + n \log n$

$a = 2$，$b = 4 \Rightarrow n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

$f(n) = n \log n$

Case 3：$f(n) = \Omega(n^{0.5+\varepsilon})$，其中$\varepsilon=0.5>0$

并且 $a f(n/b) = 2 (n/4)\log(n/4) = (1/2)n \log n - n \le (1/2)n \log n = c f(n)$，其中 $c=1/2<1$

$\therefore T(n) = \Theta(n \log n)$

例 5-14 求解递推关系 $T(n) = 2T(n/2) + n /\log n$

$a = 2$，$b = 2 \Rightarrow n^{\log_b a} = n$

$f(n) = n /\log n$

$f(n) = O(n^{\log_b a})$，但并不是多项式的比 $n^{\log_b a}$ 小，因此 Master Method 并不适用于这种情况。

从这个例子我们看到定理的 3 类情况并没有覆盖所有可能的 $f(n)$。在第一类情况和第二类情况之间有一个间隙：$f(n)$小于但不是多项式地小于 $n^{\log_b a}$；类似地，在第二类情况和第三

类情况之间也有一个间隙：$f(n)$大于但不是多项式地大于 $n^{\log_b a}$。如果函数 $f(n)$落在这两个间隙之一中，或者虽有 $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$，但附加条件 $a f(n/b) \le c f(n)$不满足，那么定理 5.1 不能适用。

5.4 分治法

对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决，否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

5.4.1 分治法的基本思想

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于 n 个元素的排序问题，当 n=1 时，不需任何计算。n=2 时，只要作一次比较即可排好序。n=3 时只要作 3 次比较即可，…。而当 n 较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。

分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。如果原问题可分割成 k ($1< k \le n$)个子问题，并且这些子问题都是可解的，进一步我们还可利用这些子问题的解求出原问题的解，那么此时使用分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。

下面我们同一个例子进一步说明分治法的基本思想。

例 5-15 寻找具有 n 个元素的数组 a[0, n-1]中的最大与最小元素。为了简化讨论，不妨设 n 为 2 的整数次幂。解答这个问题的一种直接算法是遍历数组，找出最大与最小元素，如算法 5-5。

算法 5-5 simpleMinMax

输入：整数数组 a[0, n-1]

输出：a 中的最大与最小元素

代码：

```
public IntPair simpleMinMax (int[] a){
    IntPair pair = new IntPair();
    pair.x = a[0];
    pair.y = a[0];
    for (int i=1; i<a.length; i++){
        if (pair.x<a[i]) pair.x = a[i];
        if (pair.y>a[i]) pair.y = a[i];
    }
    return pair;
}
```

```
private class IntPair{
    int x;
    int y;
}
```

算法 5-5 中一共要进行 2n-2 次比较。

然而另外一种解决这个问题的方法是使用分治法：可以把数组 a 分成大小相等的两个数组 a1 和 a2，在 a1 和 a2 中分别找出最大和最小元素，然后比较 a1 和 a2 中的最大和最小元素，两个最大元素中大的就是原数组中的最大元素，两个最小元素中小的就是原数组中的最小元素。为了要在 a1 和 a2 中找到最大与最小元素，可以重复上述过程，分别将 a1 分为 a11、a12 以及将 a2 分为 a21、a22，该过程可以一直进行下去，直到在某次分割后的数组中元素的个数少于三个的时候，此时最多只需要一次比较就可以找出该数组中的最大和最小元素。算法 5-6 表示了这个工作过程。

算法 5-6 min_max

输入：整数数组 a[0, n-1]

输出：a 中的最大与最小元素

代码：

```
public IntPair min_max(int[] a, int low, int high){// 1.
    IntPair pair = new IntPair();                  // 2.  定义见算法 5-5
    if (low>high-2){                               // 3.
        if (a[low]<a[high]){                       // 4.
            pair.x = a[high]; pair.y = a[low]; }   // 5.
        else{                                      // 6.
            pair.y = a[high]; pair.x = a[low]; }   // 7.
    }                                              // 8.
    else{                                          // 9.
        int mid = (low + high)/2;                  // 10.
        IntPair p1 = min_max(a,low,mid);           // 11.
        IntPair p2 = min_max(a,mid+1,high);        // 12.
        pair.x = p1.x>p2.x ? p1.x : p2.x;          // 13.
        pair.y = p1.y<p2.y ? p1.y : p2.y;          // 14.
    }                                              // 15.
    return pair;                                   // 16.
}                                                  // 17.
```

在算法 5-6 中为了找到 a 中的最大和最小元素：⑴ 我们使用第 10 行代码的 mid 将数组 a 分成了两个子数组。⑵ 如果子数组规模仍然较大，这里是元素个数大于 2，则在第 11、12 行代码中递归处理这两部分；如果子数组规模小于等于 2，则在 3-8 行中直接找出子数组中的最大和最小元素。⑶ 如果 a 的两个子数组中的最大和最小元素分别找到，则在 13、14 行代码中使用两个子数组中的最大和最小元素求出数组 a 中的最大最小元素。

算法 5-6 的时间复杂度可以写成一下递推关系式：

$$\begin{cases} T(2) = 1 \\ T(n) = 2T(n/2) + 2 \quad n>2 \end{cases}$$

使用递归树对该递推关系式进行分析，如图 5-3 所示。

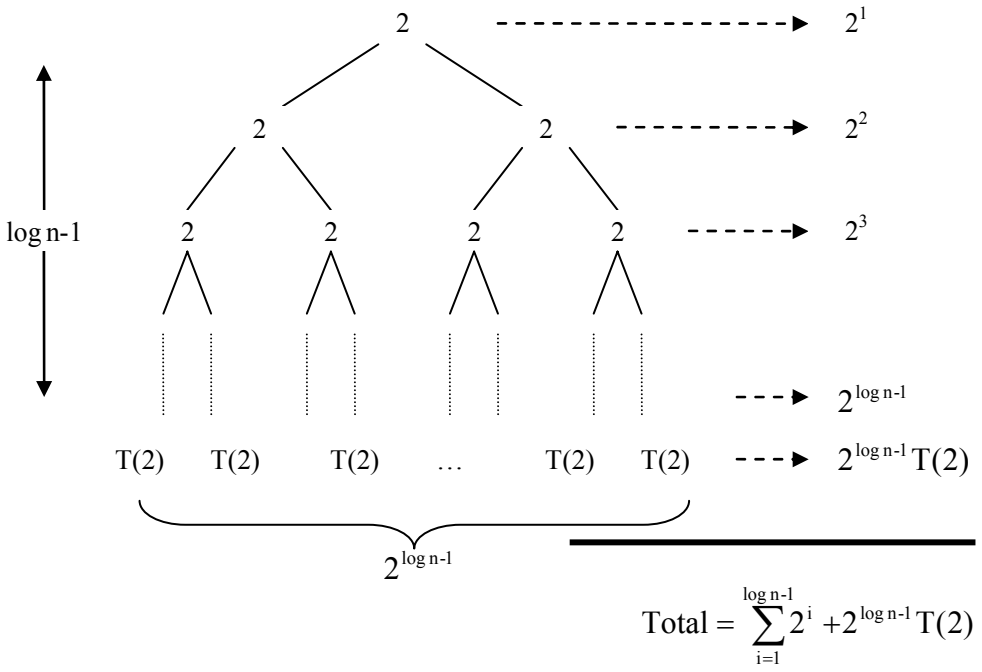


图 5-3 使用递归树分析算法 5-6

$T(n) = \sum_{i=1}^{\log n-1} 2^i + 2^{\log n-1}T(2) = n - 2 + n/2 = 3n/2 - 2$，这比使用算法 5-5 要好上许多。

通过对算法 5-6 的分析与总结，我们给出分治算法的一般设计步骤：

1. 划分步骤：在算法的这个步骤中，把输入的问题实例划分为 k≥1 个子问题，每个实例的规模严格小于问题的原始规模 n。一般来说，应尽量将 k 个子问题的规模大致相同。k=2 是最通常的情况，例如算法 5-6 中就是这样。有时也有 k=1 的划分，例如折半查找，但是这种情况等价于输入数据被分割为两部分，而其中一部分被舍弃了。当然 k 也是可以取其他值。
2. 治理步骤：当子问题的规模大于某个预定的阈值时，这个步骤是由 k 个递归调用组成的；如果子问题的规模小于阈值时则直接对问题进行求解。例如算法 5-6 的阈值是 2。
3. 组合步骤：这个步骤是组合 k 个子问题的解来得到期望的原问题的解。组合步骤对分治算法的性能起到非常关键的影响，算法的效率在很大程度上依赖于组合步骤地实现。

在下面的两小节中我们再介绍两个使用分治法求解问题的例子。

5.4.2 矩阵乘法

令 A、B 是两个 n×n 矩阵，我们希望计算它们的乘积 C = AB。下面讨论如何将分治法

运用到这个问题的求解上。

■ 传统方法

在传统方法中，我们是使用两个矩阵乘积的定义来求解 C = AB 的。C 中每个元素由以下公式计算

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \cdot B(k,j)$$

从公式中很容易看出，为计算矩阵C共需要n^3次乘法运算和n^3-n^2次加法运算。因此算法的时间复杂度$T(n) = \Theta(n^3)$。

■ 简单分治法

假设$n=2^k(k\geq 0)$，如果n>1，则A，B和C可以分成 4 个大小为n/2×n/2 的矩阵

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\ B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},\ C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

如果用分治法来计算矩阵 C，则可以进行如下计算

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

划分步骤：将矩阵 A、B 分成 4 个 n/2×n/2 的矩阵；

治理步骤：当 n>1 时，递归计算 8 个 n/2×n/2 的矩阵的乘积；

组合步骤：计算治理步骤得到 n/2×n/2 的矩阵的和。

在这里需要 8 次 n/2×n/2 矩阵乘法和 4 次 n/2×n/2 矩阵加法，由此算法的时间复杂度可以由下面的递推关系式表示

$$\begin{cases} T(1) = 1 \\ T(n) = 8T(n/2) + 4(n/2)^2 \quad n>1 \end{cases}$$

由Master Method知$T(n) = \Theta(n^3)$，可见分治法并没有产生更有效的算法，相反使用分治法它所消耗的时间比传统方法还要多，这主要是由分治法的递归调用引起的系统开销造成的。如果我们对这里分治法执行中所需的乘法与加法分开计算，得到的结果与传统方法中使用的乘法与加法的次数是一样的，实际上这里的分治法不过是传统方法的递归形式罢了，只不过是矩阵元素相乘的次序上不同而已。

下面我们将寻找更有效的分治法来解决这个问题。

■ STRASSEN 算法

STRASSEN 算法与简单的分治法之间的区别在于它使用了 7 次 n/2×n/2 矩阵乘法和 18 次 n/2×n/2 矩阵加法。

我们仍然在当 n>1 时，将 A，B 和 C 可以分成 4 个大小为 n/2×n/2 的矩阵

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\ B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

为了计算 C，我们先计算以下 7 个 n/2×n/2 矩阵的乘积

$d1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$d2 = (A_{21} + A_{22})B_{11}$

$d3 = A_{11}(B_{12} - B_{22})$

$d4 = A_{22}(B_{21} - B_{11})$

$d5 = (A_{11} + A_{12})B_{22}$

$d6 = (A_{21} - A_{11})(B_{11} + B_{12})$

$d7 = (A_{12} - A_{22})(B_{21} + B_{22})$

然后可以通过下面公式求出 C

$$C = \begin{pmatrix} d1 + d4 - d5 + d7 & d3 + d5 \\ d2 + d4 & d1 + d3 - d2 + d6 \end{pmatrix}$$

此时算法的时间复杂度可以由下面的递推关系式表示

$$\begin{cases} T(1) = 1 \\ T(n) = 7T(n/2) + 18(n/2)^2 \quad n>1 \end{cases}$$

由Master Method知$T(n) = \Theta(n^{\log 7})$，在时间上的要优于传统方法。

5.4.3 选择问题

在具有 n 个元素的有序数组 a[0, n-1]中的中项是其中间元素。即如果 n 为奇数，则中间元素是第(n+1)/2 个元素；如果 n 为偶数则选择第 n/2 这个中间元素作为中项。那么综合两种情况，中项是第⌊(n+1)/2⌋个元素。

寻找中项或任意的第 k 小元素的一个直接方法是对所有的元素排序，并取出相应元素，然而使用这种方法需要的时间较多，至少需要 Ω(n log n)时间，这是因为任何一种基于比较的排序方法在最坏的情况下都需要这么多时间。

下面我们介绍一种使用分治法在 Θ(n)时间内就可以找到中项或第 k 小项的算法。在使用分治法找出中项或第 k 小元素的基本思想是：在分治法递归调用的每一个划分步骤中都舍弃一定比例的元素，而在剩余的元素中寻找目标。于是问题的规模便以几何级数递减，例如我们假设不管处理什么对象，算法都舍弃 1/4 的元素，而对剩余的 3/4 元素进行递归，那么在第二次调用时元素的个数变为 3n/4，第三次调用时变为 9n/16，… 等等。现在假定在每次调用中，算法对每个元素的处理时间不超过常数 c，则整个算法的运行时间为：

$cn + (3/4)cn + (3/4)^2cn + \ldots + (3/4)^icn + \ldots$

这个几何级数的总和小于

$$\sum_{i=0}^{\infty} cn(3/4)^i = 4cn = \Theta(n)$$

根据上面的思想方法，可以如下设计算法来寻找数组 a 的中项或第 k 小元素：

⑴ 当$n \le n_0$时，直接对数组排序，返回第k个元素即可，否则转⑵。

⑵ 把元素划分为 p = n/5 组，每组 5 个元素，不足 5 个元素的忽略。

⑶ 取每组的中项，构成一个规模为 p 的数组 M。

⑷ 对数组 M 递归求出其中项 mm。

⑸ 使用 mm 将原数组分成 3 部分：a1 存放小于 mm 的元素，a2 存放等于 mm 的元素，

a3 存放大于 mm 的元素。

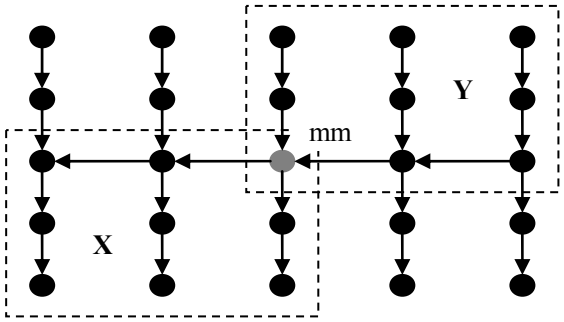
⑹ 分三种情况分别处理 a1、a2、a3

如果|a1|≥k，对 a1 递归执行算法；否则

如果|a1|+|a2|≥k，mm 是第 k 小元素，返回；否则

对 a3 递归执行算法。

下面对以上步骤进行简要说明，步骤⑵—⑷的作用主要是为了求出 p 个中项的中项 mm，参见图 5-4。



箭头由小元素指向大元素

图 5-4 中项的中项 mm 的位置

于是我们知道矩形 X 中的元素均大于等于 mm，矩形 Y 中的元素均小于等于 mm。这样在第⑸步中使用 mm 对数组 a 进行划分时，可以保证 a1 与 a3 中的元素个数大约不会小于数组 a 中元素个数的 1/4。最后导致在第⑹步中不论对 a1 或是 a3 进行递归时都可以保证至少舍弃大约 1/4 左右的元素。

以上过程的具体实现如算法 5-7。

算法 5-7 selectK

输入：整数数组 a[0, n-1]

输出：a 中的第 k 小元素

代码：

```
public int selectK(int[] a, int n, int k){
    if (n<38) {
        mergeSort (a, 0, a.length-1);  //使用归并排序[1]直接对数组a排序
        return a[k-1];
    }
    int[] m = new int[n/5];
    for (int i=0; i<n/5; i++){
        m[i] = mid(a,5*i,5*i+4);
    }
    int mm = selectK(m, m.length, (m.length+1)/2);
    int[] a1 = new int[3*n/4];
    int[] a3 = new int[3*n/4];
    int r=0,s=0,t=0;
    for (int i=0;   i<n;   i++){
        if (a[i]<mm) { a1[r++] = a[i]; continue;}
```

[1] 归并排序的方法在第十章中详细介绍。

```
            if (a[i]==mm){ s++; continue;}
            if (a[i]>mm) { a3[t++] = a[i]; continue;}
        }
        if (k<=r) return selectK (a1,r,k);
        else if (k<=r+s) return mm;
        else return selectK (a3,t,k-r-s);
    }
```

在算法中的第⑴步需要常数时间 Θ(1)，第⑵、⑶步共需要 Θ(n)时间，第⑷步需要 T(⌊n/5⌋)时间，第⑸步需要 Θ(n)时间，第⑹步所需时间分析如下：

通过图 5-4 我们知道

$|a1| \geq 3\lceil \lfloor n/5 \rfloor / 2 \rceil \geq \frac{3}{2}\lfloor n/5 \rfloor$，因此

$|a3| \leq n-|a1| \leq n - \frac{3}{2}\left(\frac{n-4}{5}\right) = 0.7n + 1.2$

由对称性得到

$|a3| \geq \frac{3}{2}\lfloor n/5 \rfloor$，$|a1| \leq 0.7n + 1.2$

如果令n_0=38，则对于所有$n>n_0$，$0.7n + 1.2 \leq \lfloor 3n/4 \rfloor$。因此在第⑹步中无论对a1 还是a3 进行递归，时间都不会超过T(⌊3n/4⌋)时间。

因此 T(n)的递推关系式为

$$T(n) \leq \begin{cases} c & n<38 \\ T(\lfloor n/5 \rfloor) + T(\lfloor 3n/4 \rfloor) + cn & n \geq 38 \end{cases}$$

解出 T(n)≤ 20cn = Θ(n)。

第六章　树

前面我们介绍了线性表、栈和队列，这些数据结构都是线性结构，在本章中我们介绍一种重要的非线性结构——树。在第二章曾经介绍，在树结构中数据元素之间的逻辑关系是前驱唯一而后续不唯一，即数据元素之间是一对多的关系。如果直观的观察，树结构是具有分支的层次结构。树结构在客观世界中广泛存在，如行政区划、社会组织机构、家族世系等都可以抽象为树结构。树结构在计算机科学领域也有非常广泛的应用，例如文件系统、编译系统、数据库系统、域名系统等领域。

本章重点讨论二叉树的存储表示及其各种运算，并研究一般树和森林与二叉树的转换关系，最后介绍树的应用实例。从本章开始逐渐将注意力转向算法，对于抽象数据类型的完整封装实现可以通过本书提供的源代码获得。

6.1 树的定义及基本术语

树是由一个集合以及在该集合上定义的一种关系构成的。集合中的元素称为树的结点，所定义的关系称为父子关系。父子关系在树的结点之间建立了一个层次结构。在这种层次结构中有一个结点具有特殊的地位，这个结点称为该树的根结点，或简称为树根。我们可以形式地给出树的递归定义如下：

树（tree）是 n（$n \geq 0$）个结点的有限集。它

1) 或者是一棵空树（$n = 0$），空树中不包含任何结点。
2) 或者是一棵非空树（$n > 0$），此时有且仅有一个特定的称为**根（root）**的结点；当$n > 1$ 时，其余结点可分为m（$m > 0$）个互不相交的有限集T_1，T_2，…，T_m，其中每一个本身又是一棵树，并且称为根的**子树（sub tree）**。

例如图 6-1（a）是一棵空树、6-1（b）是只有一个根节点的树、6-1（c）是一棵有 10 个结点的树，其中A是根，其余的结点分成 3 个不相交的集合：T_1={B,E,F}、T_2={C,G}、T_3={D,H,I,J}，每个集合都构成一棵树，且都是根A的子树。例如T_1是一棵树，其中B是根，其余结点构成 2 个不相交的集合：T_{11}={E}、T_{12}={F}是B的子树，并且都是只有一个根结点的树。

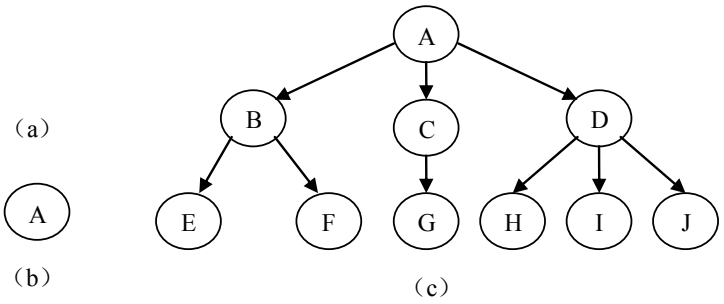


图 6-1 树的示例

下面给出树结构中的一些基本术语：

■ 结点的层次和树的深度

树的结点包含一个数据元素及若干指向其子树的若干分支。结点的**层次（level）**从根开始定义，层次数为 0 的结点是根结点，其子树的根的层次数为 1。若结点在 L 层，其子树的

根就在 L+1 层。对于层次为 k（k > 0）的每个结点 c，都有且仅有一个层次为 k-1 的结点 p 与之对应，p 称为 c 的**父亲（parent）**或父结点。若 p 是 c 的父亲，则 c 称为 p 的**孩子（child）**。父子之间的连线是树的一条边。在树中根结点没有父亲，其余结点只有一个父结点，但是却可能有多个孩子，同一结点的孩子相互称为**兄弟（sibling）**。

树中结点的最大层次数称为树的**深度（Depth）**或**高度**。树中结点也有高度，其高度是以该结点为根的树的高度。

例如，在图 6-1（c）中，结点 A 在第 0 层，结点 B、C、D 在第 1 层，结点 E、F、G、H、I、J 在第 2 层。结点 A 是结点 B、C、D 的父亲，结点 B、C、D 是结点 A 的孩子。由于结点 H、I、J 有同一个父结点 D，因此它们互为兄弟。

以 A 为根的树的高度为 2，结点 A 的高度也就为 2。

■ 结点的度与树的度

结点拥有的子树的数目称为结点的**度（Degree）**。度为 0 的结点称为**叶子（leaf）**或终端结点。度不为 0 的结点称为**非终端结点**或**分支结点**。除根之外的分支结点也称为内部结点。在这里需要注意的是结点的直接前驱结点，即它的父结点不计入其度数。

例如，在图 6-1（c）中，结点 A、D 的度为 3，结点 E、F、G、H、I、J 的度均为 0，是叶子。

在树结构中有一个重要的性质如下：

性质 6.1 树中的结点数等于树的边数加 1，也等于所有结点的度数之和加 1。

这是因为除根结点以外每个结点都与指向它的一条边对应，所以除根结点以外的结点数等于树中边数之和。因此树中的结点数等于树的边数加 1。而边数之和就是所有结点的度数之和，因此树中的结点数也等于所有结点的度数之和加 1。

性质 6.1 说明在树中结点总数与边的总数是相当的，基于这一事实，在对涉及树结构的算法复杂性进行分析时，可以用结点的数目作为规模的度量。

■ 路径

在树中k+1 个结点通过k条边连接构成的序列{（v_0,v_1），（v_1,v_2），…，（v_{k-1},v_k）| k ≥ 0}，称为长度为k的**路径（path）**。注意，此时忽略了树中边的方向。由单个结点，0 条边构成的是长度为 0 的路径。

例如，在图 6-1（c）中，{（F,B），（B,A），（A,C），（C,G）}构成了一条连接结点 F、G 长度为 4 的路径。

通过观察，不难得到如下观察结论：树中任意两个结点之间都存在唯一的路径。这意味着树既是连通的，同时又不会出现环路。从根结点开始，存在到其他任意结点的一条唯一路径，根到某个结点路径的长度，恰好是该结点的层次数。

■ 祖先、子孙、堂兄弟

将父子关系进行扩展，就可以得到祖先、子孙、堂兄弟等关系。结点的**祖先**是从根到该结点路径上的所有结点。以某结点为根的树中的任一结点都称为该结点的**子孙**。父亲在同一层次的结点互为**堂兄弟**。

例如，在图 6-1（c）中，结点 H 的祖先为结点 A、D。结点 B 的子孙有结点 E、F。结点 E、F 与结点 G、H、I、J 互为堂兄弟。

■ 有序树、m 叉树、森林

如果将树中结点的各子树看成是从左至右是有次序的，则称该树为**有序树**；若不考虑子树的顺序则称为**无序树**。对于有序树，我们可以明确的定义每个结点的第一个孩子、第二个孩子等，直到最后一个孩子。若不特别指明，一般讨论的树都是有序树。

树中所有结点最大度数为 m 的有序树称为 **m 叉树**。

森林（forest）是 m（m ≥ 0 ）棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。树和森林的概念相近。删去一棵树的根，就得到一个森林；反之，加上一个结点作树根，森林就变为一棵树。

例如，在图 6-1（c）中，以结点 A 为根的树就是一棵 3 叉树。结点 A 的所有子树可以组成一个森林。

下面给出树的抽象数据类型的定义。

ADT Tree{

数据对象 D：D 是具有相同性质的数据元素的集合。

数据关系 R：若 D=Φ 则 R =Φ；若 D≠Φ，则 R = {H}，H 是如下二元关系：

① 在 D 中存在一个唯一的称为根的元素 root，它在 H 下无前驱；

② 除 root 以外，D 中每个结点在 H 下都有且仅有一个前驱。

基本操作：

序号	方法	功能描述
(1)	getSzie ()	输入参数：无 返回参数：非负整数 功能：返回树的结点数。
(2)	getRoot()	输入参数：无 返回参数：结点 功能：返回树根结点。
(3)	getParent(x)	输入参数：结点 x 返回参数：结点 功能：返回结点 x 的父结点。
(4)	getFirstChild(x)	输入参数：结点 x 返回参数：结点 功能：返回结点 x 的第一个孩子。
(5)	getNextSibling(x)	输入参数：结点 x 返回参数：结点 功能：返回结点 x 的下一个兄弟结点，如果 x 是最后一个孩子，则返回空。
(6)	getHeight(x)	输入参数：无 返回参数：整数 功能：返回以 x 为根的树的高度。
(7)	insertChild(x,child)	输入参数：结点 x、结点 child 返回参数：无 功能：将结点 child 为根的子树插入树中，作为结点 x 的子树。
(8)	deleteChild(x,i)	输入参数：结点 x、整数 i 返回参数：无 功能：删除结点 x 的第 i 棵子树。
(9)	preOrder(x) postOrder(x) levelOrder(x)	输入参数：结点 x, 线性表 list 返回参数：迭代器 功能：先序、后序、按层遍历 x 为根的树。

}ADT Tree

6.2 二叉树

在进一步讨论树的存储结构及其操作之前，先讨论一种简单而极其重要的树结构——二叉树。因为任何树都可以转化为二叉树进行处理，并且二叉树适合计算机的存储和处理，因此在本章中二叉树是研究的重点。

6.2.1 二叉树的定义

每个结点的度均不超过 2 的有序树，称为**二叉树（binary tree）**。与树的递归定义类似，二叉树的递归定义如下：二叉树或者是一棵空树，或者是一棵由一个根结点和两棵互不相交的分别称为根的左子树和右子树的子树所组成的非空树。

由以上定义可以看出，二叉树中每个结点的孩子数只能是 0、1 或 2 个，并且每个孩子都有左右之分。位于左边的孩子称为左孩子，位于右边的孩子称为右孩子；以左孩子为根的子树称为左子树，以右孩子为根的子树称为右子树。

与树的基本操作类似，二叉树有如下基本操作：

序号	方法	功能描述
(1)	getSzie ()	输入参数：无 返回参数：非负整数 功能：返回二叉树的结点数。
(2)	isEmpty ()	输入参数：无 返回参数：boolean 功能：判断二叉树是否为空。
(3)	getRoot()	输入参数：无 返回参数：结点 功能：返回二叉树的树根结点。
(4)	getHeight()	输入参数：无 返回参数：整数 功能：返回二叉树的高度。
(5)	find(e)	输入参数：元素 e 返回参数：结点 功能：找到数据元素 e 所在结点。若 e 不存在，则返回空。
(6)	preOrder() inOrder() postOrder() levelOrder()	输入参数：无 返回参数：迭代器对象 功能：先序、中序、后序、按层遍历 x 为根的二叉树。结果由迭代器对象返回。

6.2.2 二叉树的性质

在二叉树中具有以下重要性质。

性质 6.2 在二叉树的第i层上最多有 2^i个结点。

该性质易由数学归纳法证明。证明略。

由性质 6.2 可以得到如下的进一步结论：

性质 6.3 高度为h的二叉树至多有 $2^{h+1}-1$ 个结点。

证明略。

性质 6.4 对任何一棵二叉树T，如果其终端结点数为n_0，度为 2 的结点数为n_2，则$n_0 = n_2 + 1$。

证明： 假设二叉树中结点总数为n，n_1为度为 1 的结点。

于是有：$n=n_0+n_1+n_2$

由性质 6.1 知：$n =1\times n_1+2\times n_2+1$

所以：$n_0 = n_2 + 1$

下面介绍两种特殊的二叉树，然后讨论其有关性质。

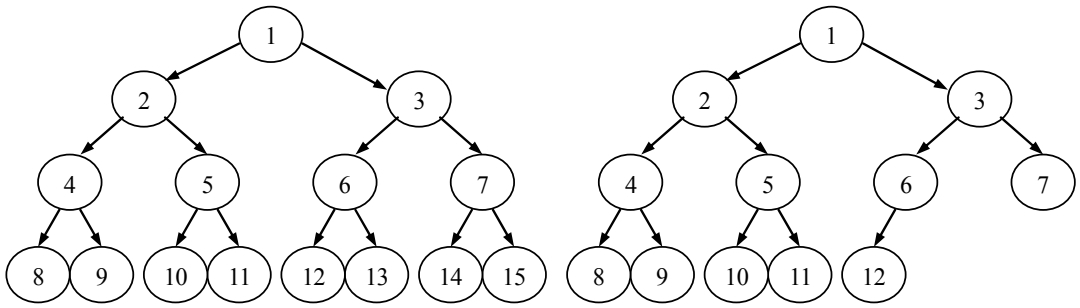
满二叉树：高度为k并且有 $2^{k+1}-1$ 个结点的二叉树。在满二叉树中，每层结点都达到最大数，即每层结点都是满的，因此称为满二叉树。图 6-2（a）所示的二叉树就是一棵满二叉树。

可以对满二叉树的结点进行编号，约定编号从根结点起，层间自上而下，层内自左而右，逐层由 1 到 n 进行标号。

完全二叉树：若在一棵满二叉树中，在最下层从最右侧起去掉相邻的若干叶子结点，得到的二叉树即为完全二叉树。

如果按照上述对满二叉树结点编号的方法，对具有 n 个结点的完全二叉树中结点进行编号，那么完全二叉树中 1~ n 号结点的位置与满二叉树中 1~ n 号结点的位置是一致的。图 6-2（b）所示的二叉树就是一棵完全二叉树。

可见，满二叉树必为完全二叉树，而完全二叉树不一定是满二叉树。



（a）满二叉树　　（b）完全二叉树

图 6-2 满二叉树与完全二叉树

性质 6.5 有 n 个结点的完全二叉树的高度为$\lfloor \log n \rfloor$。

证明： 假设高度为 h，根据性质 6.3 以及完全二叉树的定义有

$2^h-1< n \le 2^{h+1}-1$ 或 $2^h \le n < 2^{h+1}$

即 $h \le \log n < h+1$

因为 h 是整数，因此 $h = \lfloor \log n \rfloor$。

从完全二叉树的定义不难得到以下观察结论：在固定结点数目的二叉树中，完全二叉树的高度是最小的。由此可以得到二叉树的性质 6.6。

性质 6.6 含有 n≥1 个结点的二叉树的高度至多为 n-1；高度至少为$\lfloor \log n \rfloor$。

性质 6.7 如果对一棵有 n 个结点的完全二叉树的结点进行编号，则对任一结点 i($1 \le i \le n$)，有

⑴ 如果 i=1，则结点 i 是二叉树的根，无双亲；如果 i>1，则其双亲结点 PARENT(i)是结点$\lfloor i/2 \rfloor$。

⑵ 如果 2i>n，则结点 i 无左孩子；否则其左孩子是结点 2i。

⑶ 如果 2i+1>n，则结点 i 无右孩子；否则其右孩子是结点 2i+1。

证明： 先证明结论⑵、⑶。

当 i=1 时，由完全二叉树的定义知，如果 $2i = 2 \le n$，说明二叉树中存在两个或两个以上结点，所以其左孩子的编号为 2；若 2>n，说明二叉树中不存在编号为 2 的结点，因此它的左孩子不存在。同理如果 $2i+1 = 3 \le n$，说明二叉树中存在三个或三个以上结点，所以其右孩子的编号为 3；若 3>n，说明二叉树中不存在编号为 3 的结点，因此它的右孩子不存在。

当i>1 时：① 若i是第j层的第一个结点，则$i=2^j$，且其左孩子为j+1 层的第一个结点，编号为 $2^{j+1} = 2(2^j) = 2i$，如果 2i>n，则无左孩子；其右孩子为j+1 层的第二个结点，编号为 2i+1，如果 2i+1>n，则无右孩子。 ② 假设第j层上某个结点的编号为i，且 2i+1<n，则其左右孩子的编号分别为 2i和 2i+1。那么编号为i+1 的结点是i号结点的右邻，若它有左孩子，其编号为i号结点左孩子编号加 2，即 2i+2 = 2(i+1)，若它有右孩子，其编号为i号结点右孩子编号加 2，即 2i+1+2 = 2(i+1)+1。

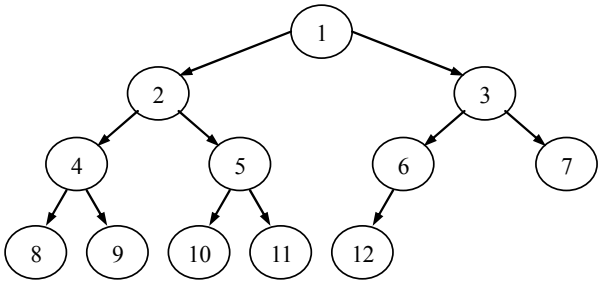
由结论⑵、⑶容易证明结论⑴。

6.2.3 二叉树的存储结构

二叉树的存储结构有两种：顺序存储结构和链式存储结构。

■ 顺序存储结构

对于满二叉树和完全二叉树来说，可以将其数据元素逐层存放到一组连续的存储单元中，如图 6-3 所示。用一维数组来实现顺序存储结构时，将二叉树中编号为 i 的结点存放到数组中的第 i 个分量中。如此根据性质 6.7，可以得到结点 i 的父结点、左右孩子结点分别存放在$\lfloor i/2 \rfloor$、2i 以及 2i+1 分量中。



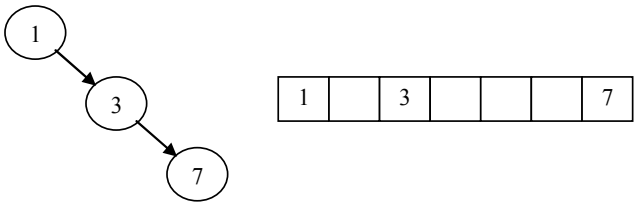
（a）完全二叉树

1	2	3	4	5	6	7	8	9	10	11	12

（b）二叉树的顺序存储结构

图 6-3 顺序存储结构

这种存储方式对于满二叉树和完全二叉树是非常合适也是高效方便的。因为满二叉树和完全二叉树采用顺序存储结构既不浪费空间，也可以根据公式很快的确定结点之间的关系。但是对于一般的二叉树而言，必须用“虚结点”将一棵二叉树补成一棵完全二叉树来存储，否则无法确定结点之间的前驱后续关系，但是这样一来就会造成空间的浪费。一种极端的情况是，为了存储k个结点，需要 2^k-1 个存储单元，图 6- 4 说明了这一情况。此时存储空间浪费巨大，这是顺序存储结构的一个缺点。

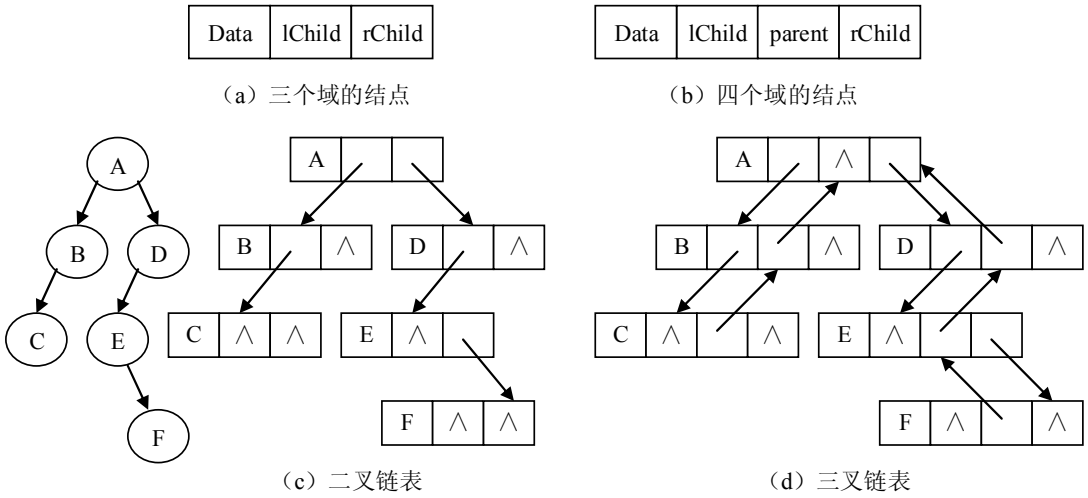


（a）单支二叉树　　（b）顺序存储结构

图 6-4 单支二叉树的顺序存储结构

■ 链式存储结构

设计不同的结点结构可构成不同的链式存储结构。在二叉树中每个结点都有两个孩子，则可以设计每个结点至少包括 3 个域：数据域、左孩子域和右孩子域。数据域存放数据元素，左孩子域存放指向左孩子结点的指针，右孩子域存放指向右孩子结点的指针。如图 6-5（a）所示。利用此结点结构得到的二叉树存储结构称为二叉链表。容易证明在具有 n 个结点的二叉链表中有 n+1 个空链域。



（a）三个域的结点　　（b）四个域的结点

（c）二叉链表　　（d）三叉链表

图 6-5 二叉树的链式存储结构

为了方便找到父结点，可以在上述结点结构中增加一个指针域，指向结点的父结点。如图 6-5（b）所示。采用此结点结构得到的二叉树存储结构称为三叉链表。在具有 n 个结点的三叉链表中也有 n+1 个空链域。

不同的存储结构实现二叉树操作的方法也不同。例如要找某个结点的父结点，在三叉链表中很容易实现；在二叉链表中则需从根结点出发一一查找。在实际应用中，要根据二叉树的主要操作来选择存储结构。

为了方便的找到父结点，我们以三叉链表作为二叉树的存储结构，并且在 6.3 节中，二叉树的基本操作的实现也是基于三叉链表来实现的。下面我们首先给出具有四个域的结点结构的定义。

代码 6-1 二叉树存储结构结点定义

```
public class BinTreeNode implements Node {
    private Object data;       //数据域
    private BinTreeNode parent;  //父结点
    private BinTreeNode lChild;  //左孩子
    private BinTreeNode rChild;  //右孩子
    private int height;          //以该结点为根的子树的高度
    private int size;            //该结点子孙数（包括结点本身）
```

```
public BinTreeNode() { this(null); }
public BinTreeNode(Object e) {
    data = e;    height = 0; size = 1;
    parent = lChild = rChild = null;
}
/******Node 接口方法******/
public Object getData() { return data; }
public void setData(Object obj) { data = obj;}

/******辅助方法,判断当前结点位置情况******/
//判断是否有父亲
public boolean hasParent(){ return parent!=null;}
//判断是否有左孩子
public boolean hasLChild(){ return lChild!=null;}
//判断是否有右孩子
public boolean hasRChild(){ return rChild!=null;}
//判断是否为叶子结点
public boolean isLeaf(){ return !hasLChild()&&!hasRChild();}
//判断是否为某结点的左孩子
public boolean isLChild(){ return (hasParent()&&this==parent.lChild);}
//判断是否为某结点的右孩子
public boolean isRChild(){ return (hasParent()&&this==parent.rChild);}

/******与 height 相关的方法******/
//取结点的高度,即以该结点为根的树的高度
public int getHeight() { return height; }
//更新当前结点及其祖先的高度
public void updateHeight(){
    int newH = 0;//新高度初始化为 0,高度等于左右子树高度加 1 中的大者
    if (hasLChild()) newH = Math.max(newH,1+getLChild().getHeight());
    if (hasRChild()) newH = Math.max(newH,1+getRChild().getHeight());
    if (newH==height) return;     //高度没有发生变化则直接返回
    height = newH;                //否则更新高度
    if (hasParent()) getParent().updateHeight();  //递归更新祖先的高度
}

/******与 size 相关的方法******/
//取以该结点为根的树的结点数
public int getSize() { return size; }
//更新当前结点及其祖先的子孙数
public void updateSize(){
    size = 1;         //初始化为 1,结点本身
    if (hasLChild()) size += getLChild().getSize();    //加上左子树规模
```

```
        if (hasRChild()) size += getRChild().getSize();    //加上右子树规模
        if (hasParent()) getParent().updateSize();         //递归更新祖先的规模
    }

    /******与 parent 相关的方法******/
    //取父结点
    public BinTreeNode getParent() { return parent; }
    //断开与父亲的关系
    public void sever(){
        if (!hasParent()) return;
        if (isLChild()) parent.lChild = null;
        else            parent.rChild = null;
        parent.updateHeight();   //更新父结点及其祖先高度
        parent.updateSize();     //更新父结点及其祖先规模
        parent = null;
    }

    /******与 lChild 相关的方法******/
    //取左孩子
    public BinTreeNode getLChild() { return lChild; }
    //设置当前结点的左孩子,返回原左孩子
    public BinTreeNode setLChild(BinTreeNode lc){
        BinTreeNode oldLC = this.lChild;
        if (hasLChild()) { lChild.sever();} //断开当前左孩子与结点的关系
        if (lc!=null){
            lc.sever();                     //断开 lc 与其父结点的关系
            this.lChild = lc;           //确定父子关系
            lc.parent = this;
            this.updateHeight();        //更新当前结点及其祖先高度
            this.updateSize();          //更新当前结点及其祖先规模
        }
        return oldLC;                   //返回原左孩子
    }

    /******与 rChild 相关的方法******/
    //取右孩子
    public BinTreeNode getRChild() { return rChild; }
    //设置当前结点的右孩子,返回原右孩子
    public BinTreeNode setRChild(BinTreeNode rc){
        BinTreeNode oldRC = this.rChild;
        if (hasRChild()) { rChild.sever();} //断开当前右孩子与结点的关系
        if (rc!=null){
            rc.sever();                     //断开 lc 与其父结点的关系
            this.rChild = rc;           //确定父子关系
```

```
                rc.parent = this;
                this.updateHeight();      //更新当前结点及其祖先高度
                this.updateSize();        //更新当前结点及其祖先规模
            }
            return oldRC;                 //返回原右孩子
        }
    }
```

代码 6-1 说明：代码中判断当前结点位置情况的辅助方法以及简单的 get 方法都在常数时间内可以完成，实现也相应非常简单。下面主要讨论 updateHeight ()、updateSize ()、sever()、setLChild(lc)、getRChild(rc)的实现与时间复杂度。

⑴ updateHeight ()：若当前结点 v 的孩子发生变化，就需要使用 updateHeight ()方法更新当前结点及其祖先结点的高度。请注意，由于一个结点的高度发生变化，会影响到其祖先结点的高度，在这里我们允许直接对任何结点执行这一操作。

因为在二叉树中任何一个结点的高度，都等于其左右子树的高度中大者加 1，而左右子树的高度只需要获取该结点左右孩子的高度即可获得，只需要 Θ(1)时间。续而从 v 出发沿 parent 引用逆行向上，依次更新各祖先结点的高度即可。如果在上述过程中，发现某个结点的高度没有发生变化，算法可以直接终止。综上所述，当对一个结点 v 调用 updateHeight () 方法时，若 v 的层数为 level(v)，则最多只需要更新 level(v)+1 个结点的高度，因此算法的时间复杂度 T(n) = O(level(v))。

⑵ updateSize ()：同样如果结点 v 的孩子发生变化，应该更新当前结点以及其祖先的规模。因为在二叉树中任何一个结点的规模，都等于其左右子树的规模之和加上结点自身，而左右子树的规模只需要获取该结点左右孩子的规模即可获得，只需要 Θ(1)时间。因此算法的时间复杂度 T(n) = O(level(v))。

⑶ sever()：切断结点 v 与父结点 p 之间的关系。该算法需要修改 v 与 p 的指针域，需要常数时间。除此之外由于 p 结点的孩子发生了变化，因此需要调用 updateHeight ()和 updateSize ()来更新父结点 p 及其祖先的高度与规模。其时间复杂度 T(n) = O(level(v))。

⑷ setLChild(lc)、getRChild(rc)：两个算法的功能相对，一个是设置结点 v 的左孩子，一个是设置结点 v 的右孩子。两个算法的实现是类似的，以 setLChild()为例说明。首先，如果 v 有左孩子 oldLC，则应当调用 oldLC. sever()断开 v 与其左孩子的关系。其次，调用 lc. sever() 断开其与父结点的关系。最后，建立 v 与 lc 之间的父子关系，并调用 v. updateSize ()与 v. updateHeight ()更新 v 及其祖先的规模与高度。

6.3 二叉树基本操作的实现

二叉树的基本操作在 6.2.1 小节中已经定义，在这些操作中有一组非常重要的操作就是遍历操作，下面首先介绍遍历及其实现，然后介绍其他操作的实现。在以下操作的实现中涉及了实现二叉树的 BinaryTreeLinked 类中定义的两个成员变量：一个是二叉树结点类型的 root 变量，它指向二叉树的根结点；另一个是第三章中定义的 Strategy 接口类型变量 strategy，用于完成数据元素之间的比较操作。

所谓树的**遍历（traversal）**，就是按照某种次序访问树中的所有结点，且每个结点恰好访问一次。也就是说，按照被访问的次序，可以得到由树中所有结点排成的一个序列。树的遍历也可以看成是人为的将非线性结构线性化。这里的“访问”是广义的，可以是对结点作各种处理，例如输出结点信息、更新结点信息等。在我们的实现中，并不真正的“访问”这

些结点，而是得到一个结点的线性序列，以线性表的形式输出。

回顾二叉树的定义，我们知道二叉树可以看成是由三个部分组成的：一个根结点、根的左子树和根的右子树。因此如果能够遍历这三部分，则可以遍历整棵二叉树。如果用 L、D、R 分别表示遍历左子树、访问根结点、遍历右子树。那么对二叉树的遍历次序就可以有 6 种方案：

⑴ 遍历左子树，访问根，遍历右子树（LDR）
⑵ 遍历左子树，遍历右子树，访问根（LRD）
⑶ 访问根，遍历左子树，遍历右子树（DLR）
⑷ 访问根，遍历右子树，遍历左子树（DRL）
⑸ 遍历右子树，遍历左子树，访问根（RLD）
⑹ 遍历右子树，访问根，遍历左子树（RDL）

在上述 6 种遍历方案中，如果规定对左子树的遍历先于对右子树的遍历，那么还剩下 3 种情况：DLR、LDR、LRD。根据对根访问的不同顺序，分别称 DLR 为先根（序）遍历，LDR 为中根（序）遍历，LRD 为后根（序）遍历。

请注意，这里的先序遍历、中序遍历、后序遍历是递归定义的，即在左右子树中也是按相应的规律进行遍历。下面给出三种遍历方法的递归定义。

⑴ 先序遍历（DLR）二叉树的操作定义为：
若二叉树为空，则空操作；否则
① 访问根结点；
② 先序遍历左子树；
③ 先序遍历右子树。

⑵ 中序遍历（LDR）二叉树的操作定义为：
若二叉树为空，则空操作；否则
① 中序遍历左子树；
② 访问根结点；
③ 中序遍历右子树。

⑶ 后序遍历（LRD）二叉树的操作定义为：
若二叉树为空，则空操作；否则
① 后序遍历左子树；
② 后序遍历右子树；
③ 访问根结点。

下面先以一棵二叉树表示一个算术表达式，然后对其进行遍历。以二叉树表示表达式的递归定义如下：若表达式为数或简单变量，则相应二叉树中仅有一个根结点；若表达式=（第一操作数）（运算符）（第二操作数），则相应二叉树用左子树表示第一操作数，用右子树表示第二操作数，根结点存放运算符。例如图 6-6（a）所示的二叉树表示下述表达式

a＋(b－c)×d－e／f

如果对该二叉树进行三种遍历，分别得到的遍历序列如下

先序遍历：－ ＋ a× － b c d ／ e f

中序遍历：a ＋ b － c×d － e ／ f

后序遍历：a b c － d× ＋ e f ／ －

从表达式上看，以上三个序列正好是表达式的前缀表示（波兰式）、中缀表示和后缀表示（逆波兰式）。在计算机中，使用后缀表达式易于求值。

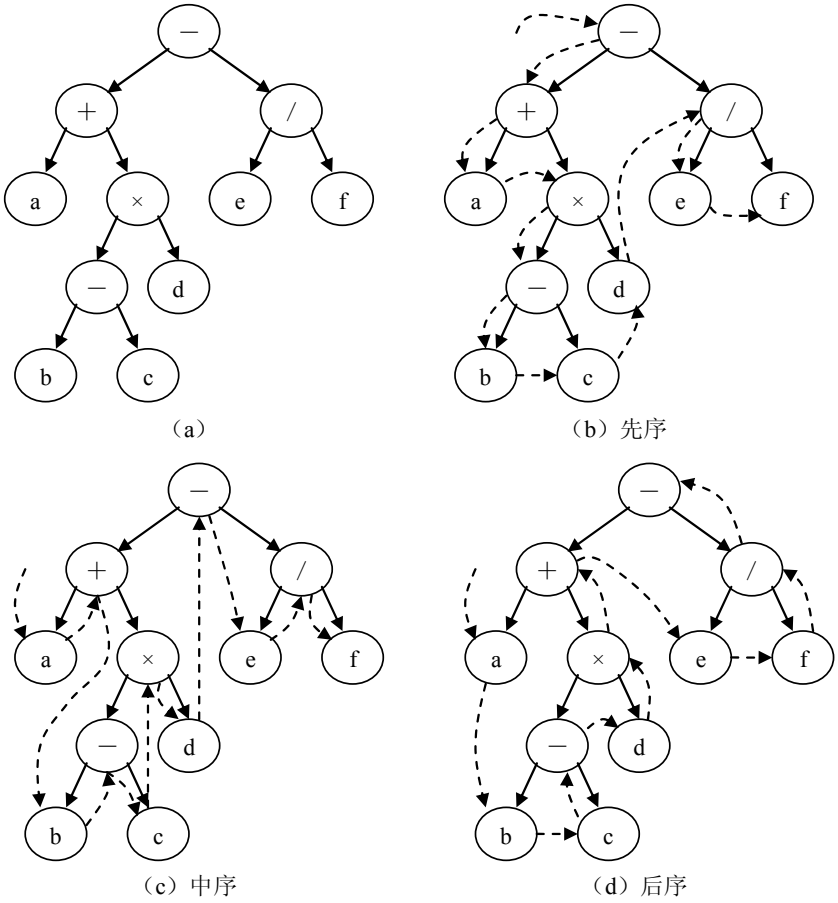


图 6-6 算术式的二叉树表示及其遍历

上述遍历的过程显然是一个递归的过程，因此可以很容易的写出三种遍历的递归程序，下面仅给出先序遍历二叉树基本操作的递归算法在三叉链表上的实现。中序遍历和后序遍历的递归算法与先序遍历的递归算法类似，这里不再一一列举。

算法 6-1 preOrder

输入：无

输出：迭代器对象，先序遍历二叉树结果

代码：

```
//先序遍历二叉树
public Iterator preOrder() {
    LinkedList list = new LinkedListDLNode();
    preOrderRecursion(this.root,list);
    return list.elements();
}
//先序遍历的递归算法
private void preOrderRecursion(BinTreeNode rt, LinkedList list){
    if (rt==null) return;                          //递归基,空树直接返回
    list.insertLast(rt);                           //访问根结点
    preOrderRecursion(rt.getLChild(),list); //遍历左子树
    preOrderRecursion(rt.getRChild(),list); //遍历右子树
}
```

算法中是将结点加入链接表 list 的尾部作为对结点的访问，该操作只需要常数时间即可完成（在随后的算法中也有同样的结论）。在算法的递归执行过程中，每个结点访问且仅被访问一次，因此算法的时间复杂度 T(n) = O(n)。对于中序和后序遍历的递归算法也是如此，即中序和后序递归算法的时间复杂度也是O(n)。

二叉树的先序、中序和后序遍历操作，其不同之处仅在于访问访问根、左子树、右子树的顺序不同而已，实则三种遍历方法的递归执行过程是一样的。图 6-7（b）中用带箭头的虚线表示了三种遍历算法的递归过程。其中，向下的箭头表示更深一层的递归调用，向上的箭头表示从递归调用推出返回。在图 6-7（b）中可以看到每个结点在遍历过程中都被途经 3 次，三种不同的遍历只是在该执行过程中的不同时机返回根结点而已。先序遍历是在第一次向下进入根结点时访问根结点，中序遍历是第二次从左子树递归调用返回时访问根，后序遍历是第三次从右子树递归调用返回时访问根。虚线旁边的①、②、③就是三种不同的访问根结点的时机，分别对应先序、中序和后序遍历。

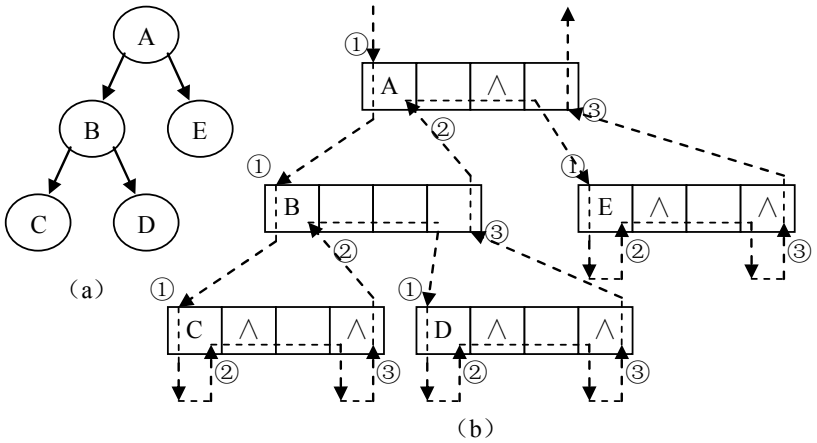


图 6-7 遍历的递归执行过程

根据上述先序、中序和后序遍历递归算法的执行过程，可以写出相应的先序、中序和后序遍历的非递归算法。

算法 6-2 preOrder

输入：无

输出：迭代器对象，先序遍历二叉树的结果

代码：

```
public Iterator preOrder() {
    LinkedList list = new LinkedListDLNode();
    preOrderTraverse (this.root,list);
    return list.elements();
}
//先序遍历的非递归算法
private void preOrderTraverse(BinTreeNode rt, LinkedList list){
    if (rt==null) return;
    BinTreeNode p = rt;
    Stack s = new StackSLinked();
    while (p!=null){
        while (p!=null){                                    //向左走到尽头
            list.insertLast(p);                             //访问根
            if (p.hasRChild()) s.push(p.getRChild());       //右子树根结点入栈
```

```
                p = p.getLChild();
            }
            if (!s.isEmpty()) p = (BinTreeNode)s.pop(); //右子树根退栈遍历右子树
        }
    }
```

算法 6-2 说明：preOrderTraverse 方法以一棵树的根结点 rt 以及链接表 list 作为参数。如果 rt 为空直接返回，否则 p 指向 rt，并先序遍历以 p 为根的树。在 preOrderTraverse 内层循环中，沿着根结点 p 一直向左走，沿途访问经过的根结点，并将这些根结点的非空右子树入栈，直到 p 为空。此时应当取出沿途最后碰到的非空右子树的根，即栈顶结点（以 p 指向），然后在外层循环中继续先序遍历这棵以 p 指向的子树。如果堆栈为空，则表示再没有的右子树需要遍历，此时结束外层循环，完成整棵树的先序遍历。如果以 rt 为根的树的结点数为 n，由于每个结点访问且仅被访问一次，并且每个结点最多入栈一次和出栈一次，因此 preOrderTraverse 的时间复杂度 T(n) = O(n)。

算法 6-3 inOrder

输入：无

输出：迭代器对象，中序遍历二叉树的结果

代码：

```
//中序遍历二叉树
public Iterator inOrder(){
    LinkedList list = new LinkedListDLNode();
    inOrderTraverse (this.root,list);
    return list.elements();
}
//中序遍历的非递归算法
private void inOrderTraverse(BinTreeNode rt, LinkedList list){
    if (rt==null) return;
    BinTreeNode p = rt;
    Stack s = new StackSLinked();
    while (p!=null||!s.isEmpty()){
        while (p!=null){          //一直向左走
            s.push(p);            //将根结点入栈
            p = p.getLChild();
        }
        if (!s.isEmpty()){
            p = (BinTreeNode)s.pop();//取出栈顶根结点访问之
            list.insertLast(p);
            p = p.getRChild();      //转向根的右子树进行遍历
        }//if
    }//out while
}
```

算法 6-3 说明：inOrderTraverse 方法以一棵树的根结点 rt 以及链接表 list 作为参数。如果 rt 为空直接返回，否则 p 指向 rt，并中序遍历以 p 为根的树。在 inOrderTraverse 内层循环中，沿着根结点 p 一直向左走，沿途将根结点入栈，直到 p 为空。此时应当取出上一层根结点访问之，然后转向该根结点的右子树进行中序遍历。如果堆栈和 p 都为空，则说明没有更

多的子树需要遍历，此时结束外层循环，完成整棵树的遍历。inOrderTraverse 的时间复杂度与 preOrderTraverse 一样 T(n) = Ｏ(n)。

算法 6-4 postOrder

输入：无

输出：迭代器对象，后序遍历二叉树的结果

代码：

```
//后序遍历二叉树
public Iterator postOrder(){
    LinkedList list = new LinkedListDLNode();
    postOrderTraverse (this.root,list);
    return list.elements();
}
//后序遍历的非递归算法
private void postOrderTraverse(BinTreeNode rt, LinkedList list){
    if (rt==null) return;
    BinTreeNode p = rt;
    Stack s = new StackSLinked();
    while(p!=null||!s.isEmpty()){
        while (p!=null){          //先左后右不断深入
            s.push(p);            //将根节点入栈
            if (p.hasLChild()) p = p.getLChild();
            else p = p.getRChild();
        }
        if (!s.isEmpty()){
            p = (BinTreeNode)s.pop();          //取出栈顶根结点访问之
            list.insertLast(p);
        }
        //满足条件时，说明栈顶根节点右子树已访问，应出栈访问之
        while (!s.isEmpty()&&((BinTreeNode)s.peek()).getRChild()==p){
            p = (BinTreeNode)s.pop();
            list.insertLast(p);
        }
        //转向栈顶根结点的右子树继续后序遍历
        if (!s.isEmpty()) p = ((BinTreeNode)s.peek()).getRChild();
        else p = null;
    }
}
```

算法 6-4 说明：postOrderTraverse 方法以一棵树的根结点 rt 以及链接表 list 作为参数。如果 rt 为空直接返回，否则 p 指向 rt，并后序遍历以 p 为根的树。在 postOrderTraverse 内层第一个 while 循环中，沿着根结点 p 先向左子树深入，如果左子树为空，则向右子树深入，沿途将根结点入栈，直到 p 为空。第一个 if 语句说明应当取出栈顶根结点访问，此时栈顶结点为叶子或无右子树的单分支结点。访问 p 之后，说明以 p 为根的子树访问完毕，判断 p 是否为其父结点的右孩子（当前栈顶即为其父结点），如果是，则说明只要访问其父亲就可以完成对以 p 的父亲为根的子树的遍历，以内层第二个 while 循环完成；如果不是，则转向

其父结点的右子树继续后序遍历。如果堆栈和 p 都为空，则说明没有更多的子树需要遍历，此时结束外层循环，完成整棵树的遍历。postOrderTraverse 的时间复杂度分析和先序、中序遍历算法一样，其时间复杂度 T(n) = Ｏ(n)。

对二叉树进行遍历的搜索路径，除了上述按先序、中序和后序外，还可以从上到下、从左到右按层进行。层次遍历可以通过一个队列来实现，算法 6-5 实现了这一操作。

算法 6-5 levelOrder

输入：无

输出：层次遍历二叉树，结果由迭代器对象输出。

代码：

```
//按层遍历二叉树
public Iterator levelOrder(){
    LinkedList list = new LinkedListDLNode();
    levelOrderTraverse(this.root,list);
    return list.elements();
}
//使用队列完成二叉树的按层遍历
private void levelOrderTraverse(BinTreeNode rt, LinkedList list){
    if (rt==null) return;
    Queue q = new QueueArray();
    q.enqueue(rt);              //根结点入队
    while (!q.isEmpty()){
        BinTreeNode p = (BinTreeNode)q.dequeue();     //取出队首结点 p 并访问
        list.insertLast(p);
        if (p.hasLChild()) q.enqueue(p.getLChild());//将 p 的非空左右孩子依次入队
        if (p.hasRChild()) q.enqueue(p.getRChild());
    }
}
```

在算法 6-5 中，每个节点依次入队一次、出队一次并访问一次，因此算法的时间复杂度 T(n) = Ｏ(n)，n 为以 rt 为根的树的结点数。

下面来分析二叉树其他基本操作的实现。由于在 BinTreeNode 中结点的高度、规模等信息已经保存，并且在发生变化时都进行了更新，因此 getSzie ()、getHeight()操作在常数时间内就能完成。isEmpty ()、getRoot()在根结点引用的基础上进行简单的比较和返回即可。树中结点的添加和删除通过结点自身的方法可以完成。唯一稍微复杂的操作是 find(e)方法的实现。算法 6-6 实现了这个操作。

算法 6-6 find

输入：元素 e

输出：e 所在结点

代码：

```
//在树中查找元素 e，返回其所在结点
public BinTreeNode find(Object e) {
    return searchE(root,e);
}
//递归查找元素 e
private BinTreeNode searchE(BinTreeNode rt, Object e) {
```

```
        if (rt==null) return null;
        if (strategy.equal(rt.getData(),e)) return rt;    //如果是根结点，返回根
        BinTreeNode v = searchE(rt.getLChild(),e); //否则在左子树中找
        if (v==null) v = searchE(rt.getRChild(),e);    //没找到，在右子树中找
        return v;
    }
```

算法 6-6 按先序顺序在二叉树中查找元素 e，并返回找到的第一个结点，如果没有找到则返回 null。算法中最多进行 n 次结点元素的比较，因此算法的时间复杂度 T(n) = O(n)。其中 strategy 为实现的二叉树的成员变量。

6.4 树、森林

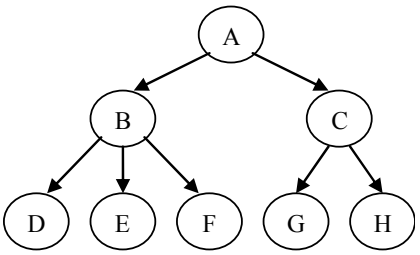
在介绍二叉树之后，我们回到树的存储及其操作的实现中来。

6.4.1 树的存储结构

树的存储结构主要有以下三种。

■ 双亲表示法

设 T 是一棵树，表示 T 的一种最简单的方法是用一个一维数组存储每个结点，数组的下标就是结点的位置指针，每个结点中有一个指向各自的父亲结点的数组下标的域。由于树中每个结点的父亲是唯一的，所以上述的父亲数组表示法可以唯一地表示任何一棵树。图 6-8 说明了这种存储结构。对于树的这种存储结构的结点结构定义，请读者自行给出。



下标	Data	Parent
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	2

图 6-8 树的双亲表示法

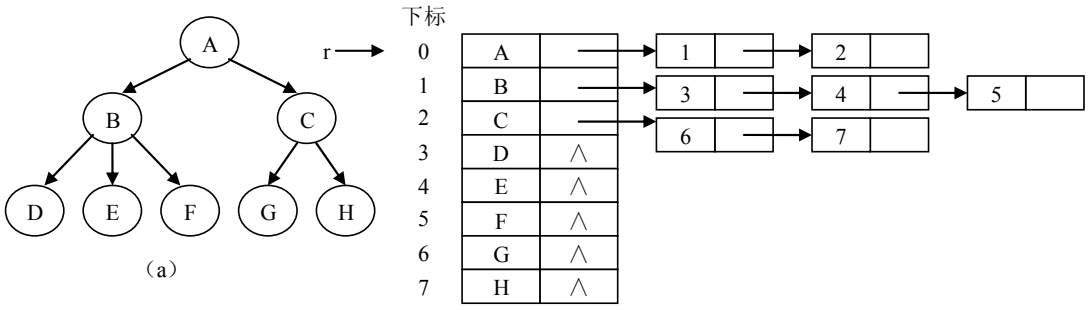
在这种表示法下，寻找一个结点的父结点只需要 O(1)时间。在树中可以从一个结点出发找出一条向上延伸到达其祖先的路径，即从一个结点到其父亲，再到其祖父等等。求这样的路径所需的时间正比于路径上结点的个数。在树的父亲数组表示法中，对于涉及查询儿子和兄弟信息的树操作，可能要遍历整个数组。为了节省查询时间，可以规定指示儿子的数组下标值大于父亲的数组下标值，而指示兄弟结点的数组下标值随着兄弟的从左到右是递增的，即如图 6-8 所示。

■ 孩子链表表示法

树的另一种常用的表示方法就是孩子链表表示法。这种表示法用一个线性表来存储树的所有结点信息，称为结点表。对每个结点建立一个孩子表。孩子表中只存储孩子结点的地址

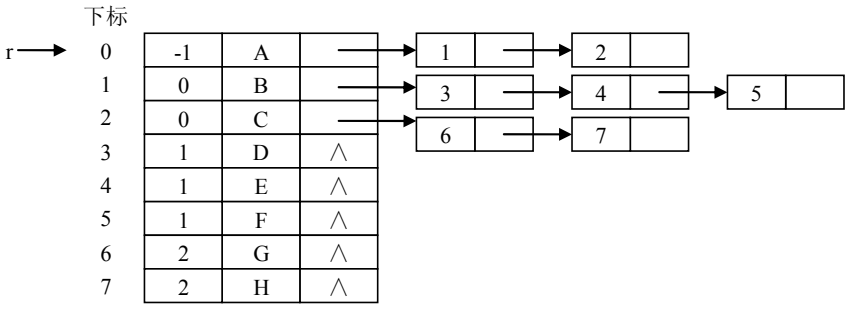
信息，可以是指针，数组下标甚至内存地址。由于每个结点的孩子数目不定，因此孩子表常用单链表来实现，因此这种表示法称为孩子链表表示法。图 6-9 是一个孩子链表表示法的示意图。

图 6-9（b）中孩子链表结构表示的树如图 6-9（a）所示，树中各结点存放于一个数组实现的表中，数组下标作为各结点的指针。每一个数组元素（即每一个结点）含有一个孩子表，在图 6-9（b）中孩子表是用单链表来实现的，当然也可以用其他表的实现方式来实现孩子表。但由于每个结点的孩子数目不确定，所以一般不用数组来实现孩子表，但可以用数组来实现结点表，就如图 6-9（b）所示。在图 6-9（b）中可以看到，位于结点表第一个位置的结点有两个孩子结点，从左到右的两个孩子结点分别位于结点表的第 2 和第 3 个位置。因为图 6-9（b）中的结点表用数组实现，所以结点的标号就是结点在结点表中的数组下标，即 1 和 2。



（a）

（b）不带双亲的孩子表示法



（c）带双亲的孩子表示法

图 6-9 树的孩子链表表示法

在孩子链表表示法中，通过某个结点找到其孩子较为容易，只需要遍历其孩子链表即可找到其所有孩子结点，然而要找到某个结点的父结点却需要对每个结点的孩子链表进行遍历，比较麻烦。因此可以在孩子链表表示法的基础上结合双亲表示法，在每个结点中再附设一个指示双亲结点的域，这样就可以在 O(1)时间内找到父结点。如图 6-9（c）所示。

■ 孩子兄弟表示法

树的孩子兄弟表示法又称为二叉树表示法。每个结点除了 data 域外，还含有两个域，分别指向该结点的第一个孩子和右邻兄弟。图 6-10 是图 6-9(a)中所示树的孩子兄弟表示法。

在图 6-10 中是使用二叉链表进行存储，这种结构便于实现树的各种操作。如果在二叉链表的每个结点中多增设一个 parent 域，则同样可以方便地实现查找父结点的操作。

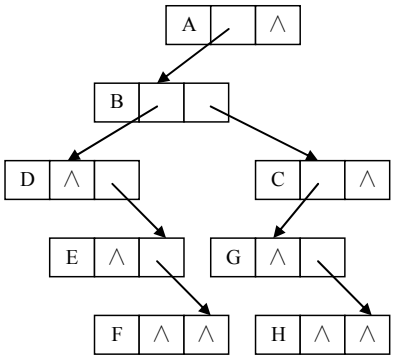


图 6-10 树的孩子兄弟表示法

6.4.2 树、森林与二叉树的相互转换

在上一小节介绍了树的存储结构，通过树的孩子兄弟表示法可以看到，树和二叉树都可以使用二叉链表作为存储结构。则以二叉链表可以导出树与二叉树之间的一个对应关系。也就是说给出一棵树可以将其唯一地对应到一棵二叉树，从实际的存储结构来看，它们的二叉链表是相同的，只是解释不同而已。图 6-11 展示了一个树与二叉树对应关系的例子。

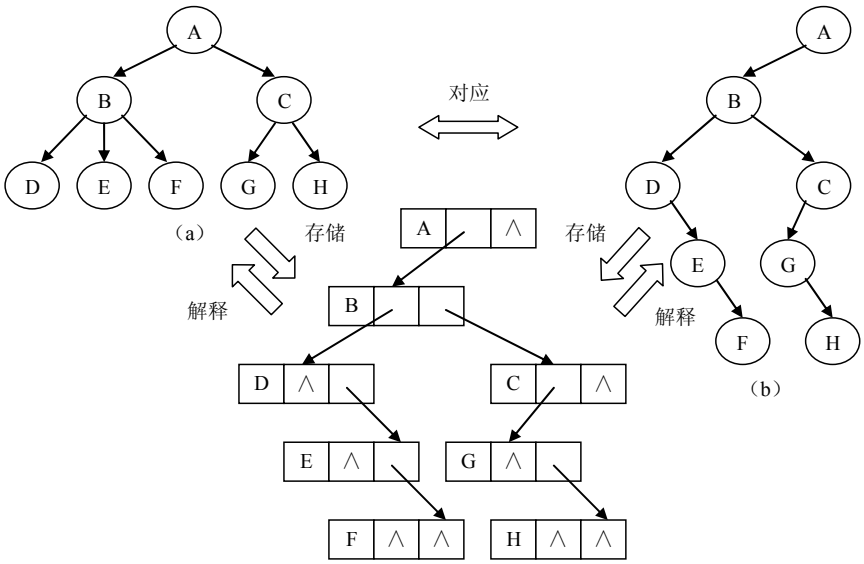


图 6-11 树与二叉树的对应关系

从树的孩子兄弟表示法的定义知道，任何一棵和某个树相对应的二叉树，其右子树必为空。下面我们给出将一棵树转换为二叉树的方法：将树中每个结点的第一个孩子转化为二叉树中对应结点的左孩子，将该结点右边的第一个兄弟转化为二叉树中该结点的右孩子。这实际上就是树的孩子兄弟表示法。

森林是若干棵树的集合。树可以转换为二叉树，森林同样也可以转换为二叉树。因此，森林也可以方便的使用孩子兄弟链表表示。森林转换为二叉树的方法是：① 将森林中的每一棵树转换为相应的二叉树。② 将所得的第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子。由以上两个步骤得到的二叉树就是森林转换得到的二叉树。

例如图 6-12 展示了森林与二叉树之间的对应关系。

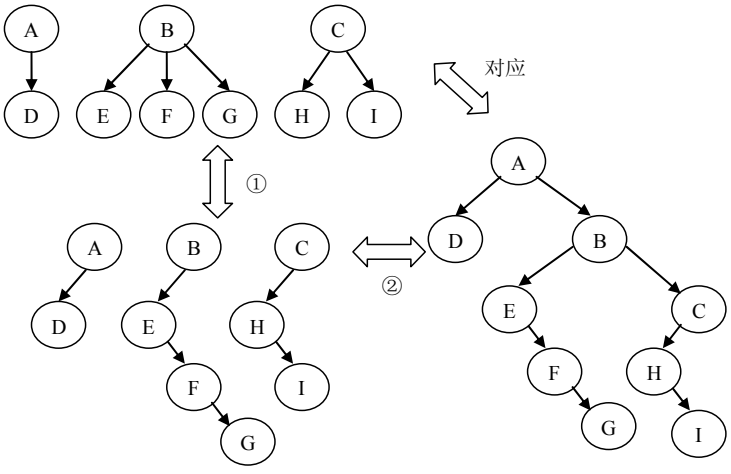


图 6-12 森林与二叉树的对应关系

森林和树都可以转换为二叉树，二者的不同是：树转换为二叉树，其根结点的右子树必然为空；而森林转换为二叉树后，其根结点有右孩子（不考虑森林退化为树和森林与树为空的情况）。将一棵二叉树还原为树或森林，具体方法如下：按层次序列对二叉树中每个结点做如下操作① 如果是根结点或者是左孩子结点，那么不做任何改动；② 如果是右孩子将其父结点设置为其当前父亲的父亲，若其当前父亲的父亲为空，则改动后其父亲为空。

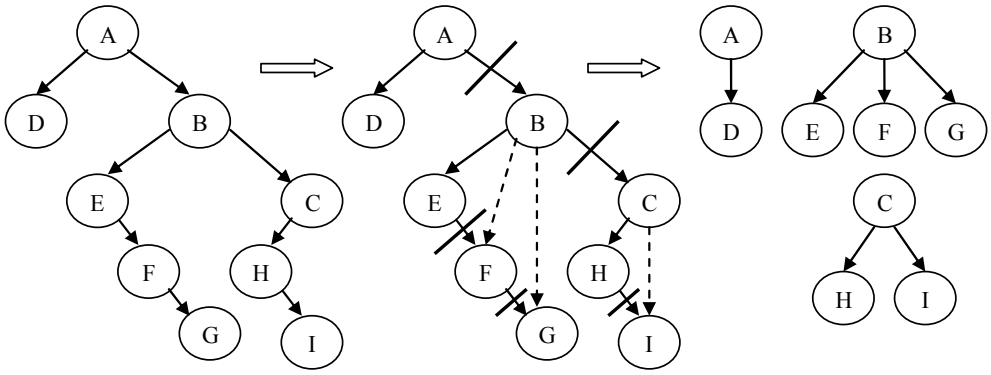


图 6-13 二叉树到森林或树的转换

这种对应关系导致森林或树与二叉树之间可以相互转换，这种相互转换可以进行递归的形式定义。

1. 森林转换成二叉树

如果F={T_1, T_2, … , T_m}是森林（m=1 时为树），则可以按照如下规则转换成一棵二叉树B=（root, LB, RB）。

① 若 F 为空，则 B 为空树。

② 若F非空，则B的根root即为森林中第一棵树的根ROOT(T_1)；B的左子树LB是从T_1中根结点的子树森林F1 转换而成的二叉树；其右子树RB是森林中除第一棵树T_1之外的森林F'={T_2, T_3, … , T_m}转换而成的二叉树。

2. 二叉树转换成森林

如果B=（root, LB, RB）是一棵二叉树，则可以按照如下规则转换成森林F={T_1, T_2, … , T_m}。

① 若 B 为空，则 F 为空。

② 若B非空，则F中第一棵树的根ROOT(T_1) 即为二叉树B的根root；T_1中根结点的子树森林F1 是由B的左子树LB是转换而成的森林；F中除第一棵树T_1之外的森林F'={T_2, T_3, … , T_m}是由B的右子树RB是转换而成的森林。

通过上述递归定义容易写出森林与二叉树相互转换的递归算法。同时森林和树的操作可以转换成二叉树的操作来实现。

6.4.3 树与森林的遍历

■ 树的遍历

由树的定义可以得到两种次序遍历树的方法：

⑴ 先根遍历

若树非空，则遍历方法为：

① 访问树的根结点，

② 从左到右，依次先根遍历根的每棵子树。

例如图 6-11（a）中树的先根遍历序列为：ABDEFCGH

⑵ 后根遍历

若树非空，则遍历方法为：

① 从左到右，依次后根遍历根的每棵子树，

② 访问树的根结点。

例如图 6-11（a）中树的后根遍历序列为：DEFBGHCA

■ **森林的遍历**

森林的遍历可以有以下三种方法：

⑴ 先序遍历

若森林非空，则：

① 访问森林中第一棵树的根结点；

② 先序遍历第一棵树中根结点的子树森林；

③ 先序遍历除去第一棵树后剩余的树构成的森林。

例如图 6-13 中森林的先序遍历序列为：ADBEFGCHI

⑵ 中序遍历

若森林非空，则：

① 中序遍历第一棵树中根结点的子树森林；

② 访问森林中第一棵树的根结点；

③ 中序遍历除去第一棵树后剩余的树构成的森林。

例如图 6-13 中森林的中序遍历序列为：DAEFGBHIC

⑶ 后序遍历

若森林非空，则：

① 后序遍历第一棵树中根结点的子树森林；

② 后序遍历除去第一棵树后剩余的树构成的森林；

③ 访问森林中第一棵树的根结点。

例如图 6-13 中森林的后序遍历序列为：DGFEIHCBA

对照二叉树与森林之间的转换方法可以发现，森林的先序、中序、后序遍历与其相对应的二叉树的先序、中序、后序遍历的结果是相同的，因此可以用相应二叉树的遍历来验证森林的遍历结果。另外树可以看成只有一棵树的森林，所以树的先根遍历和后根遍历可以分别与森林的先序遍历和中序遍历对应，因此也就可以对应为相应二叉树的先序和中序遍历。

由此可见，当以二叉链表作为存储结构时，树的先根遍历和后根遍历可以借助相应二叉树的先序遍历和中序遍历的算法实现。

6.4.4 由遍历序列还原树结构

前面我们介绍了由二叉树、树、森林等逻辑结构按照不同的次序得到相应结构的遍历序列。那么通过遍历序列是否可以还原相应的逻辑结构呢？

我们只分析二叉树的遍历序列还原为二叉树的问题。由于森林（包括树）的各种遍历可以对应为相应二叉树的遍历，如果通过遍历序列能还原为二叉树，也就可以相应的还原为森林。

首先通过二叉树的一种遍历序列是无法还原二叉树的。如果在二叉树的三种遍历序列中给出其中的两种，是否可以唯一确定一棵二叉树呢？

由先序和中序遍历序列还原二叉树：由二叉树的先序与中序序列可以唯一确定一棵二叉

树。因为，二叉树的先序遍历先访问根结点 D，其次遍历左子树 L，然后遍历右子树 R。即在先序遍历序列中，第一个结点必为根结点；而在中序遍历时，先遍历左子树 L，然后访问根结点 D，最后遍历右子树 R，因此中序遍历序列被根结点分为两部分：根结点之前的部分为左子树结点中序序列，根结点之后的为右子树结点中序序列。通过这两部分再到先序序列中找到左右子树的根结点，如此类推，便可唯一得到一棵二叉树。

例如：已知一棵二叉树的先序序列为 EBADCFHG，其中序序列为 ABCDEFGH。图 6-14 说明了还原二叉树的过程。

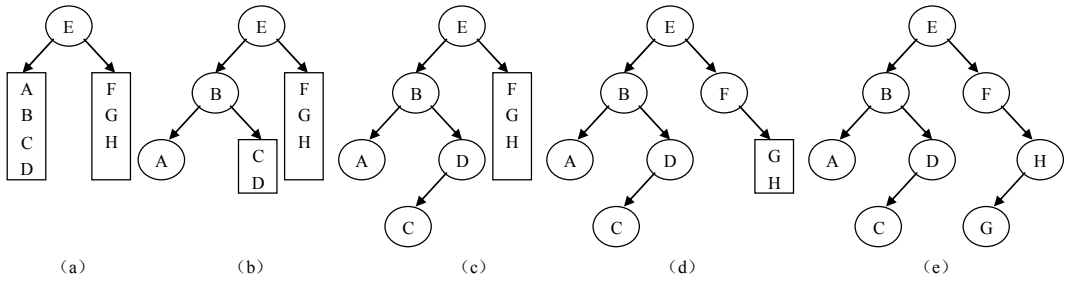


图 6-14 由先序和中序序列构造二叉树的过程

首先由先序序列知道二叉树的根结点为 E，则其左子树的中序序列为（ABCD），右子树的中序序列为（FGH）。反过来知道二叉树左子树的先序序列为（BADC），右子树先序序列为（FHG）。然后对二叉树的左右子树分别用先序和中序序列分析其根结点及其左右子树，直到得到整个二叉树结构。

由后序和中序遍历序列还原二叉树：同样，同过二叉树的后序和中序序列也可以唯一确定一棵二叉树。其方法与上述方法类似，只不过此时根结点是出现在后序序列的最后面。

由先序和后序序列不能唯一确定一棵二叉树。例如：先序序列为 AB，后序序列为 BA。此时就无法确定二叉树的结构，因为 B 既可以是根 A 的左子树，也可以是根 A 的右子树。

6.5 Huffman 树

Huffman 树又称最优树，可以用来构造最优编码，用于信息传输、数据压缩等方面，是一类有着广泛应用的二叉树。

6.5.1 二叉编码树

在计算机系统中，符号数据在处理之前首先需要对符号进行二进制编码。例如，在计算机中使用的英文字符的 ASCII 编码就是 8 位二进制编码，由于 ASCII 码使用固定长度的二进制位表示字符，因此 ASCII 码是一种定长编码。

为了缩短数据编码长度，可以采用不定长编码。其基本思想是：给使用频度较高的字符编较短的编码，这是数据压缩技术的最基本思想。如何给数据中的字符编以不定长编码，而使数据编码的平均长度最短呢？

首先分析第一个问题：如何对字符集进行不定长编码。

在一个编码系统中，任何一个编码都不是其他编码的前缀，则称该编码系统的编码是前缀码。例如：01, 10, 110, 111, 101 就不是前缀编码，因为 10 是 101 的前缀，如果去掉 10 或 101 就是前缀编码。当在一个编码系统中采用定长编码时，可以不需要分隔符；如果采用不定长编码时，必须使用前缀编码或分隔符，否则在解码时会产生歧义。所谓解码就是由二进制位串还原字符数据的过程。而使用分隔符会加大编码长度，因此一般采用前缀编码。例

6-1 说明了这个问题。

例 6-1 假设字符集为{A, B, C, D}，原文为 ABACCDA。

一种等长编码方案为 A:00　B:01　C:10　D:11，此时编解码不会产生歧义，过程如下。

编码：ABACCDA → 00010010101100

解码：00010010101100 → ABACCDA

一种不等长编码方案为：A:0　B:00　C:1　D:01，由于此编码不是前缀码，此时在编解码的过程中会产生歧义。对于同一编码可以有不同的解码，过程如下。

编码：ABACCDA → 000011010

解码：000011010 → AAAACCDA

000011010 → BBCCDA　　错误！出现歧义。

为产生没有歧义的前缀编码，可以使用二叉编码树来实现。使用二叉树对字符集中的字符进行编码的方法是，将字符集中的所有字符作为二叉树的叶子结点；在二叉树中，每一个“父亲—左孩子”关系对应一位二进制位 0，每一个“父亲—右孩子”关系对应一位二进制位 1；于是从根结点通往每个叶子结点的路径，就对应于相应字符的二进制编码。每个字符编码的长度 L 等于对应路径的长度，也等于该叶子结点的层次数。例如对于例 6-1 中的每个字符可以按照图 6-15 所示的二叉编码树进行编码。

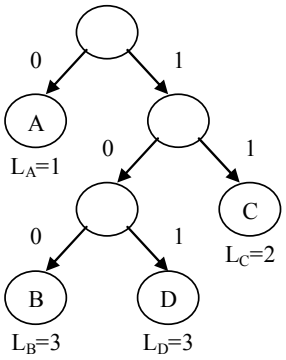


图 6-15 二叉编码树示例

按照图 6-15 中的二叉编码树对 A、B、C、D 四个字符进行编码，则 A 的编码是 0，B 的编码是 100，C 的编码是 11，D 的编码是 101。这个编码显然是一个前缀编码。

由于在二叉树中任何一个叶子结点都不会出现在根到其他叶子结点的路径上，那么按照上述二叉编码树的编码方法，任何一个叶子结点表示的编码都不会是任何其他叶子表示编码的前缀，因此由二叉编码树得到的编码都是前缀码。

反过来如果要进行解码，也可以由二叉编码树便捷的完成。解码的过程是从头开始扫描二进制编码位串，并从二叉编码树的根结点开始，根据比特位不断进入下一层结点，当碰到 0 时向左深入，为 1 时向右深入；到达叶子结点后输出其对应的字符，然后重新回到根结点，并继续扫描二进制位串直到完毕。

还是如图 6-15 所示，此时将 ABACCDA 进行编码得到：0100011111010。解码过程是从左到右扫描二进制位串。在读出最前端的 0 后，相应的从根结点到达结点，于是输出 A，重新回到根结点；依次扫描后续二进制位 100，到达叶子结点 B，于是输出 B，重新回到根结点；读出下一个二进制位 0，输出 A；读出 11，输出 C；读出 11，输出 C；读出 101，输出 D；最后读出 0，输出 A；此时二进制位串扫描完毕，相应的解码工作也完成，最后得到字符数据 ABACCDA。

6.5.2 Huffman 树及 Huffman 编码

在上一小节中介绍了如何对字符集进行不定长编码的方法，但是同时我们看到对于同一个字符集进行编码的二叉编码树可以有很多，只要叶子结点个数与字符个数对应即可。例如对例 6-1 中字符即进行编码的二叉树就可以有，但不限于图 6-16 所示的二叉树。

在这些不同的编码中哪个才是使得编码长度最小的呢？

例如在例 6-1 中，选择图 6-15 中的编码方案比选择图 6-16 中的两种编码方案好。由于字符 A、B、C、D 分别出现了 3 次、1 次、2 次、1 次。使用图 6-15 的编码方案，编码的长

度为 3×1+1×3+2×2+1×3=13；使用图 6-16（a）的编码方案，编码的长度为 3×3+1×2+2×3+1×1=18；使用图 6-16（b）的编码方案，编码的长度为 3×3+1×2+2×1+1×3=16。

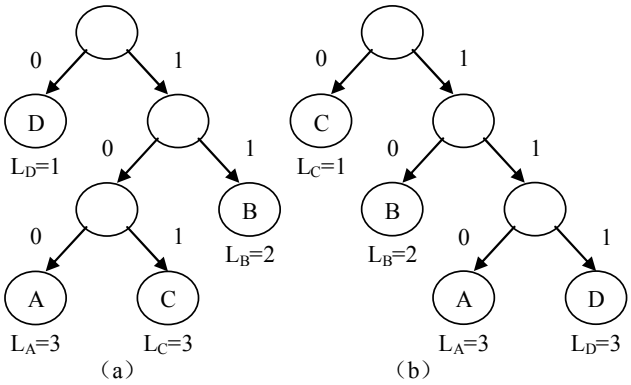


图 6-16 同一字符集的不同二叉编码树

字符集中各种字符出现的概率是不同的，字符的出现概率决定了编码方案的选择。

若假设字符 c 的出现概率为 P(c)≥0，$\sum_{c\in\Sigma} P(c)=1$，其中 Σ 为字符集合；并且将字符 c 在二叉编码树中对应叶子的层次数计为 level(c)，则每个字符 c∈Σ 的带权编码长度为 level(c)×P(c)，Σ 中所有字符的带权编码长度之和 $\sum_{c\in\Sigma} P(c)\times level(c)$ 称为二叉编码树的带权编码长度。

假设字符数据的长度为 Len，那么字符数据的编码长度 CL= $\sum_{c\in\Sigma} Len\times P(c)\times level(c)$，因此二叉编码树的带权编码长度是衡量一个编码优劣的重要指标，它决定了编码的优劣。要求编码长度最小的编码方案实际上就是寻找带权编码长度最小的二叉编码树。下面将这个问题进一步进行抽象，可以将字符的带权编码长度以及二叉编码树的带权编码长度抽象为一般二叉树中的概念。

给树中的节点赋予一个具有某种意义的正数，我们称之为该**结点的权**。结点的**带权路径长度**是从根结点到该结点之间的路径长度与结点权的乘积。**树的带权路径长度**定义为树中所有叶子结点的带权路径长度之和

$$WPL = \Sigma W_i L_i \qquad 1\le i\le n$$

其中：n为叶子结点个数，W_i为第i个叶子结点的权，L_i为从根到第i个叶子结点路径的长度。

当引入以上概念以后，求最佳编码方案实际上就抽象为求在叶子结点个数与权确定时带权路径长度最小的二叉树。那么什么样的树带权路径长度最小呢？

对于给定n个权值$w_1, w_2, \dots w_n$（n≥2），求一棵具有n个叶子结点的二叉树，使其带权路径长度$\Sigma W_i L_i$最小。由于Huffman给出了构造具有这种树的方法，因此这种树称为Huffman树。

Huffman 树：它是由 n 个带权叶子结点构成的所有二叉树中带权路径长度最小的二叉树，Huffman 树又称**最优二叉树**。

构造 Huffman 树的算法步骤如下：

① 根据给定的 n 个权值，构造 n 棵只有一个根结点的二叉树， n 个权值分别是这些二叉树根结点的权，F 是由这 n 棵二叉树构成的集合；

② 在 F 中选取两棵根结点树值最小的树作为左、右子树，构造一颗新的二叉树，置新二叉树根的权值=左子树根结点权值+右子树根结点权值；

③ 从 F 中删除这两颗树，并将新树加入 F；

④ 重复②、③，直到 F 中只含一棵树为止。

直观地看，先选择权值小的，所以权值小的结点被放置在树的较深层，而权值较大的离根较近，这样自然在 Huffman 树中权越大的叶子离根越近，这样一来，在计算树的带权路径长度时，自然会具有最小的带权路径长度，这种生成算法就是一种典型的贪心算法。

用上述算法可以验证在例 6-1 中，图 6-15 所示的二叉树就是 Huffman 树，即例 6-1 中原文 ABACCDA 的最短编码长度为 13。

例 6-2 假设有一组权值{7, 4, 2, 9, 15, 5}，试构造以这些权值为叶子的 Huffman 树。

构造 Huffman 树的过程如图 6-17 所示

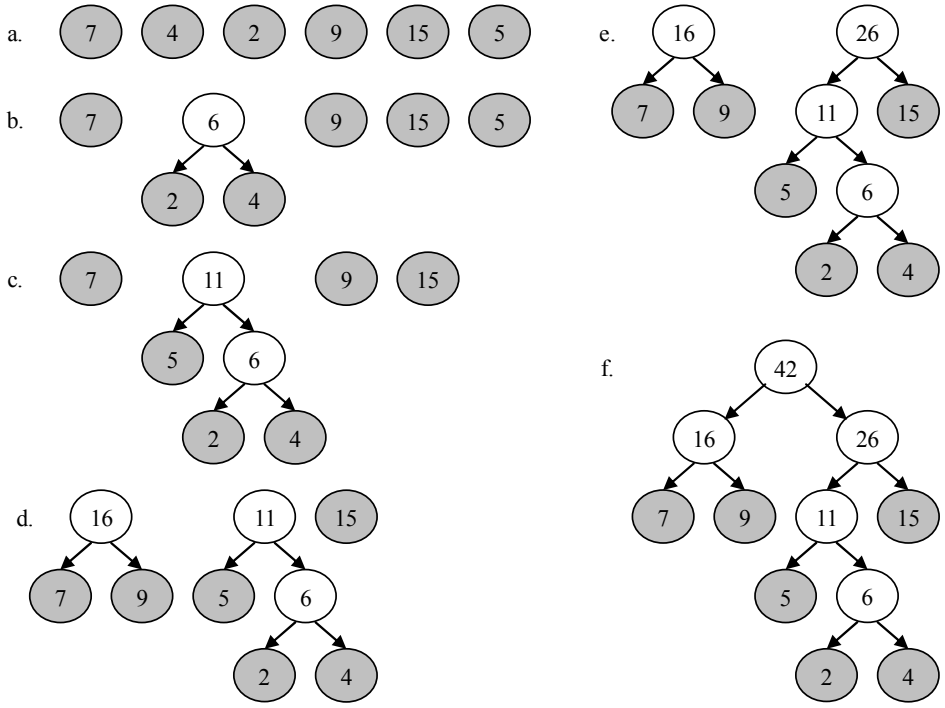


图 6-17 Huffman 树构造过程

使用二叉编码树进行编码，以字符出现的概率作为相应叶子的权值，当这棵二叉编码树是 Huffman 树时，所得到的编码称之为 **Huffman 编码**。例 6-1 中 A、B、C、D 四个字符的 Huffman 编码分别是 0、100、11、101。

下面讨论构造 Huffman 树的具体实现。

Huffman 树的实现可以使用顺序的存储结构也可以使用链式的存储结构。前面给出了二叉树的链式存储实现，这里我们也给出 Huffman 树的链式存储结构的实现。

Huffman 树也是一棵二叉树，其结点可以继承二叉树的结点来实现，但是需要两个新的属性，即权值和编码。代码 6-2 定义了 Huffman 树的节点结构。

代码 6-2 Huffman 树结点定义

```
public class HuffmanTreeNode extends BinTreeNode {
    private int weight;              //权值
    private String coding = "";      //编码
    //构造方法
    public HuffmanTreeNode(int weight){  this(weight,null);}
    public HuffmanTreeNode(int weight, Object e){
        super(e);
        this.weight = weight;
```

```
    }
    //改写父类方法
    public HuffmanTreeNode getParent() {
        return (HuffmanTreeNode)super.getParent();
    }
    public HuffmanTreeNode getLChild() {
        return (HuffmanTreeNode)super.getLChild();
    }
    public HuffmanTreeNode getRChild() {
        return (HuffmanTreeNode)super.getRChild();
    }
    //get&set 方法
    public int getWeight(){ return weight;}
    public String getCoding(){ return coding;}
    public void setCoding(String coding){ this.coding = coding;}
}
```

构造 Huffman 树的过程可以通过算法 6-7 实现。

算法 6-7 buildHuffmanTree

输入：结点数组 nodes

输出：Huffman 树的根结点

代码：

```
    //通过结点数组生成 Huffman 树
    private static HuffmanTreeNode buildHuffmanTree(HuffmanTreeNode[] nodes){
        int n = nodes.length;
        if (n<2) return nodes[0];
        List l = new ListArray();//根结点线性表，按 weight 从大到小有序
        for (int i=0; i<n; i++)          //将结点逐一插入线性表
            insertToList(l,nodes[i]);
        for (int i=1; i<n; i++){   //选择 weight 最小的两棵树合并，循环 n-1 次
            HuffmanTreeNode min1 = (HuffmanTreeNode)l.remove(l.getSize()-1);
            HuffmanTreeNode min2 = (HuffmanTreeNode)l.remove(l.getSize()-1);
            HuffmanTreeNode newRoot =
                    new HuffmanTreeNode(min1.getWeight()+min2.getWeight());
            newRoot.setLChild(min1);    newRoot.setRChild(min2); //合并
            insertToList(l,newRoot);//新树插入线性表
        }
        return (HuffmanTreeNode)l.get(0);//返回 Huffman 树的根
    }
    //将结点按照 weight 从大到小的顺序插入线性表
    private static void insertToList(List l, HuffmanTreeNode node){
        for (int j=0; j<l.getSize(); j++)
            if (node.getWeight()>((HuffmanTreeNode)l.get(j)).getWeight()){
                l.insert(j,node);
                return;
```

```
            }
        l.insert(l.getSize(),node);
    }
```

算法 6-7 说明：算法使用一个线性表l保存在生成Huffman树过程中森林F的所有树的根结点，并保持在线性表中这些根结点的权值从大到小有序。不难知道当线性表采用数组实现时方法insertToList的运行时间为O(n)。因此初始化将n个叶子结点插入线性表的时间为$O(n^2)$。在有线性表l之后，取得最小权值的 2 个根结点，只需要O(1)的时间，合并 2 棵树需要O(1)时间，将新树插入线性表l需要O(n)时间，循环执行n-1 次，因此构造Huffman树的时间为$O(n^2)$。综上所述，算法buildHuffmanTree的时间复杂度$T(n)= O(n^2)$。

Huffman 编码可以在 Huffman 树中递归生成，算法 6-8 实现了这个操作。

算法 6-8 generateHuffmanCode

输入：Huffman 树根结点

输出：生成 Huffman 编码

代码：

```
    //递归生成 Huffman 编码
    private static void generateHuffmanCode(HuffmanTreeNode root){
        if (root==null) return;
        if (root.hasParent()){
            if (root.isLChild())
                root.setCoding(root.getParent().getCoding() + "0");     //向左为 0
            else
                root.setCoding(root.getParent().getCoding() + "1");     //向右为 1
        }
        generateHuffmanCode(root.getLChild());
        generateHuffmanCode(root.getRChild());
    }
```

第七章　图

图是一种较线性结构和树结构更为复杂的数据结构，在图结构中数据元素之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。由此，图的应用也极为广泛，在诸如系统工程、控制论、人工智能、计算机网络等许多领域中，都将图作为解决问题的数学手段之一。在离散数学中主要侧重于图的理论研究，在本章中主要是讨论图在计算机中的表示，以及使用图解决一些实际问题的算法实现。

4.4 图的定义

4.4.1 图及基本术语

图（graph）是一种网状数据结构，图是由非空的顶点集合和一个描述顶点之间关系的集合组成。其形式化的定义如下：

Graph = (V , E)

V = {x| x∈某个数据对象}

E = {<u , v>| P(u , v)∧(u,v∈V)}

V 是具有相同特性的数据元素的集合，V 中的数据元素通常称为**顶点（Vertex）**，R 是两个顶点之间关系的集合。P(u , v)表示 u 和 v 之间有特定的关联属性。

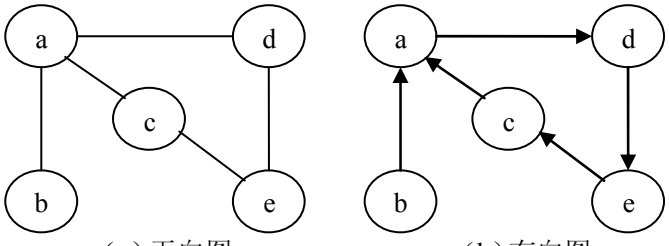
若<u , v>∈E，则<u , v>表示从顶点 u 到顶点 v 的一条弧，并称 u 为弧尾或起始点，称 v 为弧头或终止点，此时图中的顶点之间的连线是有方向的，这样的图称为**有向图（directed graph）**。

若<u , v>∈E 则必有<v , u>∈E，即关系 E 是对称的，此时可以使用一个无序对(u , v)来代替两个有序对，它表示顶点 u 和顶点 v 之间的一条边，此时图中顶点之间的连线是没有方向的，这种图称为**无向图（undirected graph）**。

在无向图和有向图中 V 中的元素都称为顶点，而顶点之间的关系却有不同的称谓，即弧或边，本章中有些内容是既涉及无向图也涉及有向图的，因此在描述图中顶点之间的关系时，分别称为弧或边较为麻烦，我们统一的将它们称为**边（edge）**。并且我们还约定顶点集与边集都是有限的，并记顶点与边的数量为**|V|**和**|E|**。

通过图的以上形式化定义，我们看到本章所讨论的“图”，并非通常所指的图形、图像或数学上的函数图。

图 7-1 分别给出了一个无向图和有向图的示例。



（a）无向图　　（b）有向图

图 7-1 图的示例

■ 简单图

图中所有的边不见得就是构成一个集合，准确地说它们构成一个复集——允许出现重复元素的集合。例如，若在某对定点之间有多条无向边，就属于这种情况，此时的图也可以含有实际的意义，比如用顶点表示城市，用边表示城市之间的航线，则有可能在一对城市之间存在多条航线。另外在图中还有一种特殊情况：某条边的两个顶点是同一个顶点。

不过，以上特殊情况并不多见。不含上述特殊边的图，称为简单图。对简单图而言，图中所有的边自然构成一个集合，并且每条边的两个顶点均不相同。在本章中所讨论的图均是简单图。

■ 邻接点

对于无向图 G = (V , E)，如果边(u , v) ∈E，则称顶点 u 与顶点 v 互为**邻接点**。边(u , v)依附于顶点 u 和 v，或者说边(u , v)与顶点 u 和 v 相**关联**。

对于有向图 G = (V , E)，如果边<u , v> ∈E，则称定点 u 邻接到顶点 v，顶点 v 邻接自顶点 u，或称 v 为 u 的邻接点，u 为 v 的逆邻接点。同样我们称边<u , v>与顶点 u 和 v 相关联。从顶点 u 出发的边也称为 u 的出边或邻接边，而指向顶点 u 的边也称为 u 的入边或逆邻接边。

■ 顶点的度、入度、出度

顶点的**度（degree）**是指依附于某顶点 v 的边数，通常记为 TD (v)。

在有向图中，要区别顶点的入度与出度的概念。顶点 v 的**入度（in degree）**是指以顶点为终点的边的数目，记为 ID (v)；顶点 v **出度（out degree）**是指以顶点 v 为起始点的边的数目，记为 OD (v)。对于有向图有 TD(v) = ID(v) ＋ OD(v)。在无向图中每条边都可以看成出边，也可以看成入边，此时 TD(v) = ID(v) = OD(v)。

例如在图 7-1（a）所示的无向图中 TD(a) = 3，TD(c) = TD(d) = TD(e) = 2，TD(b) = 1。而在图 7-1（b）所示的有向图中 ID(a) = 2，ID(c) = ID(d) = ID(e) = 1，ID(b) = 0；OD(a) = OD(b) = OD(c) = OD(d) = OD(e) = 1；TD(a) = 3，TD(c) = TD(d) = TD(e) = 2，TD(b) = 1。

通过观察可以有以下观察结论。对于任何无向图G = (V , E)，都有 $\Sigma TD(v_i) = 2|E|$，其中$v_i \in V$；因为在无向图中计算各点度数之和时，每条边都恰好被统计了两次。另外对于任何有向图G = (V , E)，都有$\Sigma ID(v_i) = \Sigma OD(v_i) = |E|$，其中$v \in V$；这是因为在计算各个顶点的出（入）度的过程中，每条有向边都只被统计了一次。由此对于有向图而言$TD(v_i) = ID(v_i) + OD(v_i) = 2|E|$。通过以上分析，我们有以下结论：

观察结论 7.1 在任何图G = (V , E)中，$|E| = (\Sigma TD(v_i))/2$。

■ 完全图 、稠密图、稀疏图

假设图中顶点个数为 n，边数为 m。

在无向图中当每个顶点都与其余 n-1 个顶点邻接时，图的边数达到最大，此时图中每两个顶点之间都存在一条无向边，边数 m 为 n 个顶点任意取出 2 个的组合数，即 m = n(n-1)/2。

同样有向图中当每个顶点都有 n-1 条出边并有 n-1 条入边时，图中边数达到最大，此时图中每两个顶点之间都存在方向不同的两条边，边数 e 为在 n 个顶点中任意取出 2 个并进行排列的排列组合数，即 m = n(n-1)。

观察结论 7.2 假设在图 G = (V , E)中有 n 个顶点和 m 条边。

1） 若 G 是无向图，则有 $0 \le m \le n(n-1)/2$。

2） 若 G 是有向图，则有 $0 \le m \le n(n-1)$。

由此，在具有n个顶点的图中，边的数目为 $O(n^2)$。由于图中边数与顶点数并非线性关系，因此在对有关图的算法时间复杂度、空间复杂度进行分析时，我们往往以图中的顶点数和边

数作为问题的规模。

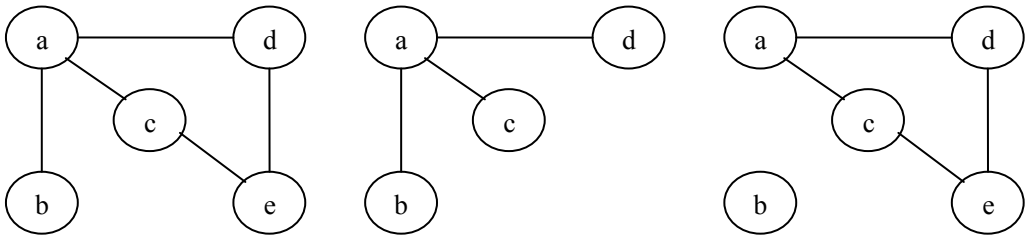
有 n(n-1)/2 条边的无向图称为**无向完全图**；有 n(n-1)条边的有向图称为**有向完全图**。有很少边（如 m < n log n）的图称为**稀疏图**，反之边较多的图称为**稠密图**。

■ 子图

设图 G = (V , E)和图 G' = (V' , E')。

如果 V'⊆ V 且 E'⊆ E，则称 G'是 G 的一个**子图（subgraph）**。以图 7-1（a）为例，若 V' = { a , b , c , d }且 E' = {(a , b) , (a , c) , (a , d)}，则 G' = (V' , E')就是图 G 的子图。

如果 V'= V 且 E'⊆ E，则称 G'是 G 的一个**生成子图（spanning subgraph）**。图 7-2 显示了子图与生成子图的示例。



（a）原图　　（b）子图　　（c）生成子图

图 7-2 子图与生成子图

■ 路径、环路及可达分量

所谓图中的一条**通路**或**路径（path）**，就是由m+1 个顶点与m条边交替构成的一个序列**ρ** = { $v_0, e_1 , v_1 , e_2 , v_2 , \dots , e_m , v_m$}，$m \geq 0$，且$e_i = (v_{i-1} , v_i)$，$1 \leq i \leq m$。路径上边的数目称为路径长度，计作|**ρ**|。

长度|**ρ**| ≥ 1 的路径，若路径的第一个顶点与最后一个顶点相同，则称之为**环路**或**环(cycle)**。

如果组成路径**ρ**的所有顶点各不相同，则称之为简单路径（simple path）；如果在组成环的所有顶点中，除首尾顶点外均各不相同，则称该环为简单环路（simple cycle）。如果组成路径**ρ**的所有边都是有向边，且e_i均是从v_{i-1}指向v_i，$1 \leq i \leq m$，则称**ρ**为有向路径，同样可以定义有向环路。

在描述简单图的路径或环路时，我们只需要依次给出组成路径或环路的各个顶点，而不必再给出具体的边。例如在图 7-1（b）中{a , d , e , c}是一条简单有向通路，而{d , e , c , a , d}是一条简单有向环路。

在有向图 G 中，若从顶点 s 到顶点 v 有一条通路，则称 v 是从 s 可达的。对于顶点 s，从 s 可达的所有顶点所组成的集合，称作 s 在 G 中对应的**可达分量**。例如在图 7-1（b）中顶点 a 的可达分量为顶点集{a , d , e , c}。

■ 连通性与连通分量

在无向图中，如果从一个顶点v_i到另一个顶点v_j($i \neq j$)有路径，则称顶点v_i和v_j是**连通**的。如果图中任意两顶点v_i、$v_j \in V$，v_i和v_j都是连通的，则称该图是**连通图（connected graph）**。例如，图 7-2（a）中的图是连通图；而图 7-2（c）中的图是非连通图，但该图有两个连通分量。所谓**连通分量（connected component）**，是指无向图的极大连通子图。显然任何连通图的连通分量只有一个，即本身。而非连通图有多个连通分量，各个连通分量之间是分离的，没有任何边相连。

在有向图中，若图中任意一对顶点v_i和v_j ($i \neq j$)均有一条从顶点v_i到另一个顶点v_j的路径，也有从v_j到v_i的路径，则称该有向图是**强连通图**。有向图的极大强连通子图称为**强连通分量**。显然任何强连通图的强连通分量只有一个，即本身。而非强连通图有多个强连通分量，各个

强连通分量内部的任意顶点之间是互通的，在各个强连通分量之间可能有边也可能没有边存在。例如图 7-3（a）中的图是非强连通图，它有两个强连通分量，如图 7-3（b）所示；如果在图 7-3（a）的图中添加一条有向边<b , e>，则可以得到一个强连通图，如图 7-3（c）所示。

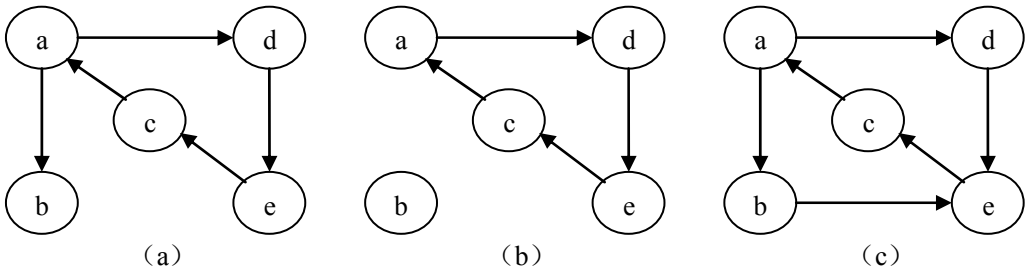


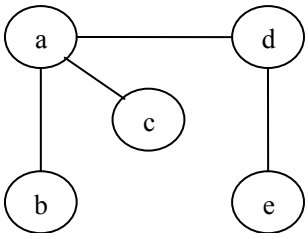
图 7-3 强连通图与强连通分量

■ 无向图的生成树

对于无向图 G = (V , E)。如果 G 是连通图，则 G 的**生成树（spanning tree）**，是 G 的一个极小连通生成子图。

图 G 的生成树必定包含图 G 的全部 n 个顶点，以及足以构成一棵树的 n-1 条边。图 7-2（a）中图 G 的生成树如图 7-4 所示。在生成树中添加任意一条属于原图中的边必定会产生回路，因为生成树本身是连通的，新添加的边使其所依附的两个顶点之间有了第二条路径。若生成树中减少任意一条边，则必然成为非连通的，因为生成树是极小连通生成子图。

一棵有 n 个顶点的生成树有且仅有 n-1 条边。如果一个图有 n 个顶点和小于 n-1 条边，则是非连通图。如果它有多于 n-1 条边，则一定有环路，不是极小连通生成子图。但是，有 n-1 条边的生成子图不一定是生成树。例如图 7-2（c）中的图有 n-1 = 4 条边，但却不是图 7-2（a）中图 G 的生成树。



7-2（a）的生成树

图 7-4 无向图的生成树

如果在生成树中确定某个顶点作为根结点，则生成树就可以成为我们在第六章中介绍的树结构。

■ 权与网

在实际应用中，图不但需要表示元素之间是否存在某种关系，而且图的边往往与具有一定实际意义的数有关，即每条边都有与它相关的实数，称为**权**。这些权值可以表示从一个顶点到另一个顶点的距离或消耗等信息，在本章中假设边的权均为正数。这种边上具有权值的图称为**带权图（weighted graph）**或**网（network）**。图 7-5 中的图就是带权图。

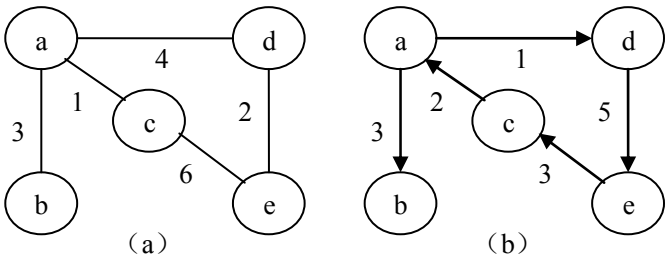


图 7-5 带权图

4.4.2 抽象数据类型

与其他数据结构一样，在介绍图的存储结构之前，先给出图的抽象数据类型和 Java 接口。在这里与前面介绍的数据结构不同的是，图有无向图和有向图之分，有些操作是无向图支持的，例如我们只求无向图的最小生成树；而有些操作是只有有向图才支持的，例如拓扑排序和求关键路径。

下面给出图的抽象数据类型定义。

ADT Graph{

数据对象 D：D 是具有相同性质的数据元素的集合。

数据关系 R：R = {<u , v>| P(u , v)∧(u,v∈D)}

基本操作：

序号	方法	功能描述
(1)	getType()	输入参数：无 返回参数：整数，图的类型。 功能：返回当前图的类型。
(2)	getVexNum() getEdgeNum()	输入参数：无 返回参数：整数。 功能：返回图中顶点数。返回图中边数。
(3)	getVertex() getEdge()	输入参数：无 返回参数：迭代器 功能：返回图中所有顶点的迭代器。返回图中所有边的迭代器。
(4)	remove(v) remove(e)	输入参数：顶点 v。边 e。 返回参数：无 功能：在图中删除特定的顶点 v。在图中删除特定的边。
(5)	insert(v) insert(e)	输入参数：顶点 v。边 e。 返回参数：无 功能：在图的顶点集中添加一个新顶点。在图的边集中添加一条新边。
(6)	areAdjacent(u, v)	输入参数：顶点 u、v 返回参数：boolean 功能：判断顶点 v 是否为顶点 u 的邻接顶点。
(7)	edgeFromTo(u, v)	输入参数：顶点 u、v 返回参数：边 功能：返回从顶点 u 到顶点 v 的边，如果不存在返回空。
(8)	adjVertexs(u)	输入参数：顶点 u 返回参数：迭代器 功能：返回顶点 u 的所有邻接点。
(9)	DFSTraverse(v)	输入参数：顶点 v 返回参数：迭代器 功能：从顶点 v 开始深度优先搜索遍历图。
(10)	BFSTraverse(v)	输入参数：顶点 v 返回参数：迭代器 功能：从顶点 v 开始广度优先搜索遍历图。

⑾	shortestPath(v)	输入参数：顶点 v 返回参数：迭代器 功能：求顶点 v 到图中所有顶点的最短路径。
⑿	generateMST()	输入参数：无 返回参数：无 功能：求无向图的最小生成树。有向图不支持此操作。
⒀	toplogicalSort()	输入参数：无 返回参数：迭代器 功能：求有向图的拓扑序列。无向图不支持此操作。
⒁	criticalPath()	输入参数：无 返回参数：无 功能：求有向无环图的关键路径。无向图不支持此操作。

}ADT Graph

对应于上述抽象数据类型，下面给出图的 Java 接口。

代码 7-1 图的接口定义

```
public interface Graph {
    public static final int UndirectedGraph = 0;//无向图
    public static final int DirectedGraph    = 1;//有向图
    //返回图的类型
    public int getType();
    //返回图的顶点数
    public int getVexNum();
    //返回图的边数
    public int getEdgeNum();
    //返回图的所有顶点
    public Iterator getVertex();
    //返回图的所有边
    public Iterator getEdge();
    //删除一个顶点 v
    public void remove(Vertex v);
    //删除一条边 e
    public void remove(Edge e);
    //添加一个顶点 v
    public Node insert(Vertex v);
    //添加一条边 e
    public Node insert(Edge e);
    //判断顶点 u、v 是否邻接，即是否有边从 u 到 v
    public boolean areAdjacent(Vertex u, Vertex v);
    //返回从 u 指向 v 的边，不存在则返回 null
    public Edge edgeFromTo(Vertex u, Vertex v);
    //返回从 u 出发可以直接到达的邻接顶点
    public Iterator adjVertexs(Vertex u);
    //对图进行深度优先遍历
    public Iterator DFSTraverse(Vertex v);
```

```
    //对图进行广度优先遍历
    public Iterator BFSTraverse(Vertex v);
    //求顶点 v 到其他顶点的最短路径
    public Iterator shortestPath(Vertex v);
    //求无向图的最小生成树,如果是有向图不支持此操作
    public void generateMST() throws UnsupportedOperation;
    //求有向图的拓扑序列,无向图不支持此操作
    public Iterator toplogicalSort() throws UnsupportedOperation;
    //求有向无环图的关键路径,无向图不支持此操作
    public void criticalPath() throws UnsupportedOperation;
}
```

其中 UnsupportedOperation 是调用图不支持的操作时抛出的异常，定义如下：

代码 7-2 UnsupportedOperation 异常

```
public class UnsupportedOperation extends RuntimeException {
    public UnsupportedOperation(String err) {
        super(err);
    }
}
```

4.5 图的存储方法

在介绍图的存储结构之前，先明确一个概念，即“顶点在图中的位置”。从图的逻辑结构定义来看，无法将图中的顶点排列成为一个唯一的线性序列。在图中，可以将任何一个顶点看成是图的第一个顶点。同理，对于任何一个顶点而言，它的邻接点之间也不存在顺序关系。但为了对图的存储和操作能更加方便，需要将图中的顶点按任意序列排列起来（该排列顺序完全是人为规定的）。所谓“顶点在图中的位置”就是指该顶点在人为确定的序列中的位置。同理，也可以对某个顶点的邻接点进行人为的排序，在这个序列中自然的形成了第 i 个邻接点的概念。

由于图的结构比较复杂，任意两个顶点之间都可能存在联系，因此无法以数据元素在存储区的位置来表示元素之间的关系，即图没有顺序映像的存储结构，但可以借助数组来表示数据元素之间的关系。

4.5.1 邻接矩阵

图的**邻接矩阵（adjacent matrix）**表示法是使用数组来存储图结构的方法，也被称为**数组表示法**。它采用两个数组来表示图：一个是用于存储所有顶点信息的一维数组，另一个是用于存储图中顶点之间关联关系的二维数组，这个关联关系数组也被称为邻接矩阵。

假设图G＝(V ，E)有n个顶点，即V＝{v_0,v_1,…,v_{n-1}}，则表示G中各顶点关联关系的为一个n×n的矩阵A，矩阵的元素为：

$$A[i,j]\begin{cases}1 & <u,v>\text{或}(u,v)\in E\\ \infty & \text{反之}\end{cases}$$

图 7-6 中两个图的邻接矩阵分别为：

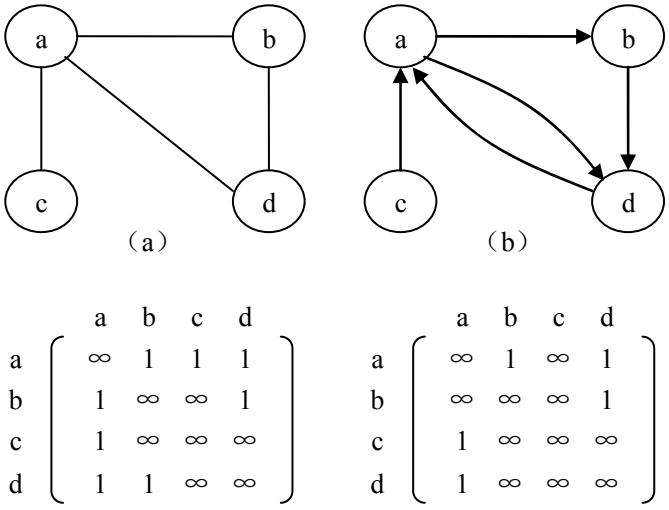


图 7-6 使用邻接矩阵存储图

并且，此时顶点 a、b、c、d 在存储顶点的数组中所对应的下标分别为 0、1、2、3。

实际上这一表示形式也可以推广至带权图，若 G 是一个有 n 个顶点的带权图，则它的邻接矩阵是具有如下性质的 n×n 的矩阵 A：

$$A[i,j]\begin{cases} W_{ij} & <u,v>\text{或}(u,v)\in E \\ \infty & \text{反之} \end{cases}$$

图 7-7 给出了一个有向带权图和它的邻接矩阵。

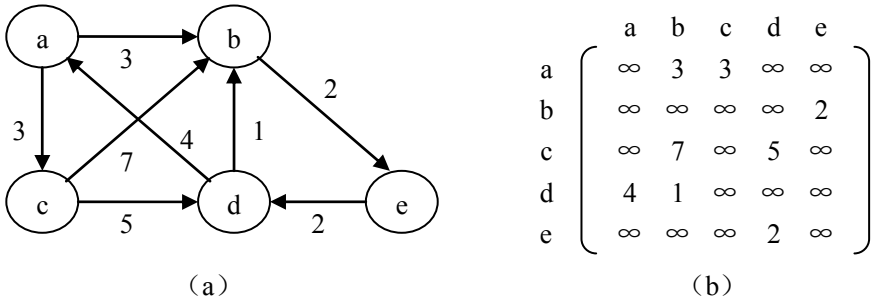


图 7-7 带权图及邻接矩阵

从图的邻接矩阵存储方法容易看出：首先，无向图的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上（或下）三角矩阵的元素即可。其次，对于无向图，邻接矩阵的第i行（或第i列）非∞元素的个数正好是第i个顶点的度$TD(v_i)$。再次，对于有向图，邻接矩阵的第i行（第i列）非∞元素的个数正好是第i个顶点的出度$OD(v_i)$（入度$ID(v_i)$）。最后，通过邻接矩阵很容易确定图中任意两个顶点之间是否有边相连；但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。

从空间上看，不论顶点u、v之间是否有边，在邻接矩阵中都需预留存储空间，因为每条边所需的存储空间为常数，所以邻接矩阵需要占用$\Theta(n^2)$的空间，这一空间效率较低。具体来说，邻接矩阵的不足主要在两个方面。首先，尽管由n个顶点构成的图中最多可以有n^2条边，但是在大多数情况下，边的数目远远达不到这个量级，因此，在邻接矩阵中大多数单元都是闲置的。其次，矩阵结构是静态的，其大小N需要预先估计，然后创建N×N的矩阵。然而，图的规模往往是动态变化的，N的估计过大会造成更多的空间浪费，如果估计过小则经常会出现空间不够用的情况。

4.5.2 邻接表

由上面的分析可知，邻接矩阵的空间效率之所以低，是因为其中大量的单元所对应的边有可能并未在图中出现，这是静态数组结构不可避免的问题。既然如此，则可以将静态的存储结构改为动态的链式存储结构。按照这一思路可以得到图的另一种表示形式，即邻接表。

邻接表（adjacency list）是图的一种链式存储方法，邻接表表示法类似于树的孩子链表表示法。在邻接表中对于图G中的每个顶点v_i建立一个单链表，将所有邻接于v_i的顶点v_j链成一个单链表，并在表头附设一个表头结点，这个单链表就称为顶点v_i的邻接表。

在邻接表中共有两种结点结构，分别是边表结点和表头结点。每个边表结点由 3 个域组成，如图 7-8（a）所示。其中邻接点域（adjvex）指示与顶点v_i邻接的顶点在图中的位置，链域（nextedge）指向下一条边所在的结点，数据域（info）存储和边有关的信息，如权值等信息。在头结点中，结构如图 7-8（b）所示，除了设有链域（firstedge）指向链表中的第一个结点之外，还有用于存储顶点v_i相关信息的数据域（data）。

图 7-8 邻接表结点结构

这些表头结点（可以链接在一起）以顺序的结构形式进行存储，以便随机访问任一顶点的链表。图 7-9 给出了图的邻接表存储示例。

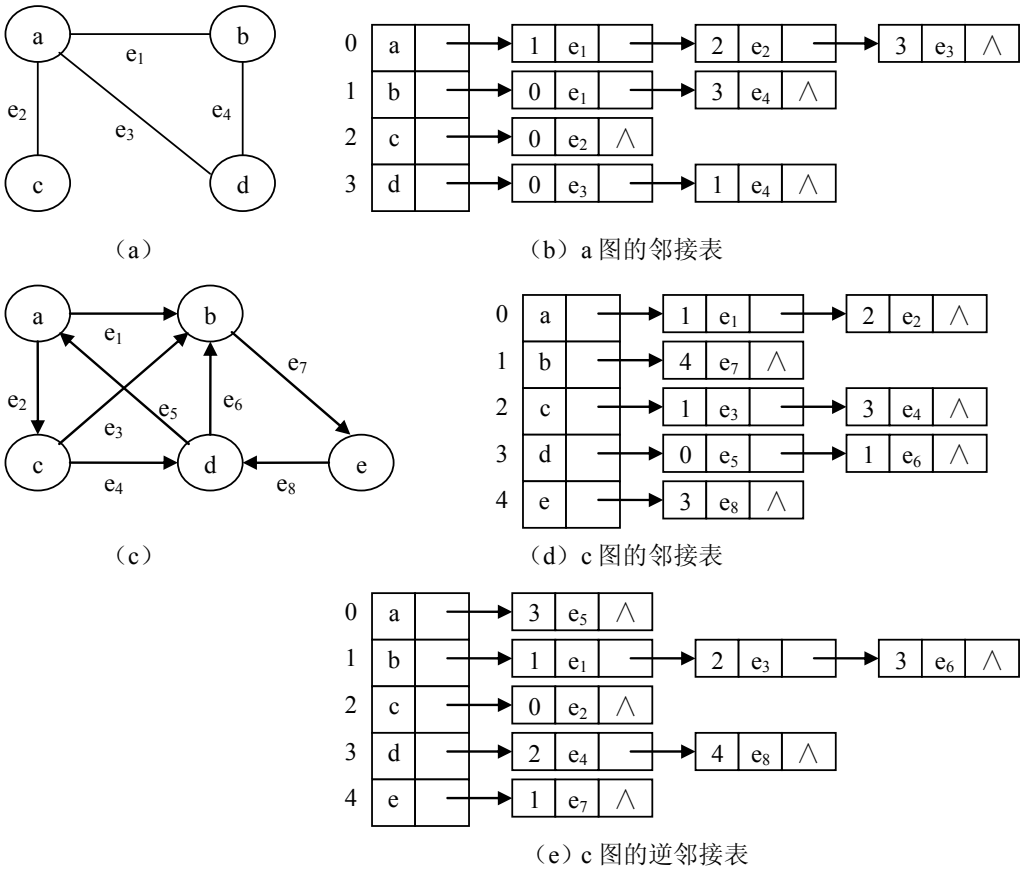


图 7-9 图的邻接表

就存储空间而言，对于 n 个顶点、m 条边的无向图，若采用邻接表作为存储结构，则需要 n 个表头结点和 2m 个边表结点。显然在边稀疏（m<<n(n-1)/2）的情况下，用邻接表存储要比使用邻接矩阵节省空间。

在无向图的邻接表中，顶点v_i的度恰为顶点v_i的邻接表中边表结点的个数；而在有向图中，顶点v_i的邻接表中边表结点的个数仅为顶点v_i的出度，为求顶点v_i的入度必须遍历整个邻接表。在所有链表中其邻接点域的值指向v_i的位置的结点个数是顶点v_i的入度。为了方便求得有向图中顶点的入度，可以建立一个有向图的逆邻接表，如图 7-9（e）所示。

在邻接表中容易找到一个顶点的邻接点，但是要判定两个顶点v_i和v_j之间是否有边，则需要搜索顶点v_i或顶点v_j的邻接表，与邻接矩阵相比不如邻接矩阵方便。

4.5.3 双链式存储结构

虽然邻接表是图的一种很有效的存储结构，在邻接表中容易求得顶点和边的各种信息。但是这种结构会给图的某些操作带来不便。例如，在无向图中，每条边在邻接表中对应了两个边表结点，如果在图的应用中需要对边进行标记，或删除边等，此时需要找到表示同一条边的两个边表结点，然后执行相同的操作，以保证数据的一致性，因此操作的实现比较麻烦。

另一方面，如果在邻接表中，将所有的顶点按照顺序的方式存储，会使得顶点的删除操作所需的时间代价较大。首先在数组中删除一个元素，平均需要移动大约数组中一半的元素；其次，在删除一个顶点时，需要将与之相关联的所有边删除，如上所述，在无向图中删除一条边需要删除两个边表结点，较为复杂；再次，由于在删除某个顶点以后，会造成后续顶点在顶点数组中的位置发生变化，因此要判断所有边表结点的邻接点域是否需要修改，如果其邻接点域所指顶点位置发生变化，则需要使用新的指向替换原来的指向。以上操作总共需要 O(|V|+|E|)的时间。解决这个问题的一种办法是，在删除顶点时，并不将数组中其后续顶点前移，只是将相应位置设置为空，然后删除与之关联的所有边。但是这种方法会使得在图中添加顶点之前需要先遍历顶点数组，查找数组中为空的位置，如果有则将新的顶点放入该位置，如果没有则放到数组的尾部。这样添加一个新顶点的操作实现会比较复杂。

综合以上两点，在图的邻接表与逆邻接表的基础上，我们给出图的一种双链式存储结构以解决上述问题。首先在双链式存储结构中，我们不再以邻接表中的边表结点表示一条边，而是将图中的顶点和边都抽象成为一个独立的类，使用顶点对象表示图中的顶点，使用边对象表示图中的边。其次，所有的顶点都存储在一个链接表中，而不是使用数组来存储；并且所有的边也存储在一个链接表中。图的双链式存储结构如图 7-10 所示。

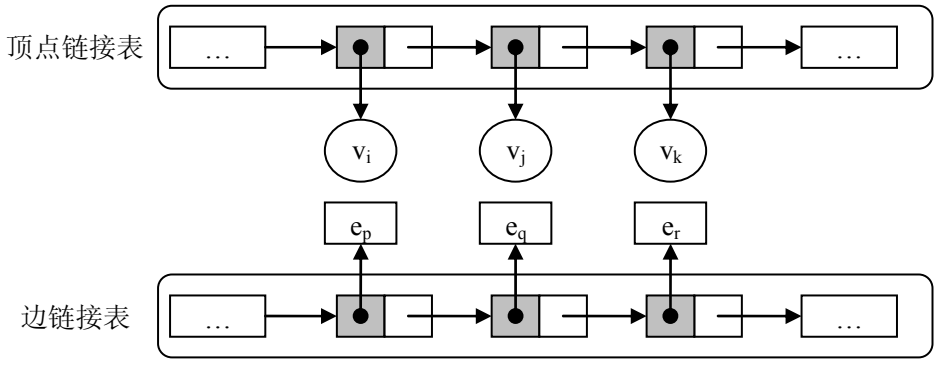


图 7-10 图的双链式存储结构

当然这只是双链式结构的初步模型，为了完整的实现图结构并能方便的实现图的操作，我们还需要给出顶点和边的详细结构，并在顶点与边之间建立联系。下面以一条有向边 e = <u , v>和与之关联的两个顶点 u、v 为例来说明图中顶点和边的结构以及它们之间的联系。

顶点、边的结构以及它们之间的联系如图 7-11 所示。

在顶点中有 3 个重要的指针域：顶点位置域、邻接边域、逆邻接边域。其中顶点位置域指向顶点在顶点链接表中所在的结点，以此可以在 O(1)时间内确定顶点在图中的位置。在无向图中顶点的邻接边域指向的链接表存储了与该顶点关联的所有边的引用，顶点的逆邻接边域为空；而在有向图中，顶点的邻接边域指向的链接表存储了该顶点所有出边的引用，顶点的逆邻接边域指向的链接表存储了该顶点所有入边的引用。邻接边域和逆邻接边域相当于图中顶点的邻接表和逆邻接表。通过这两个域可以很快的找到与该顶点相连的所有顶点和边的信息。

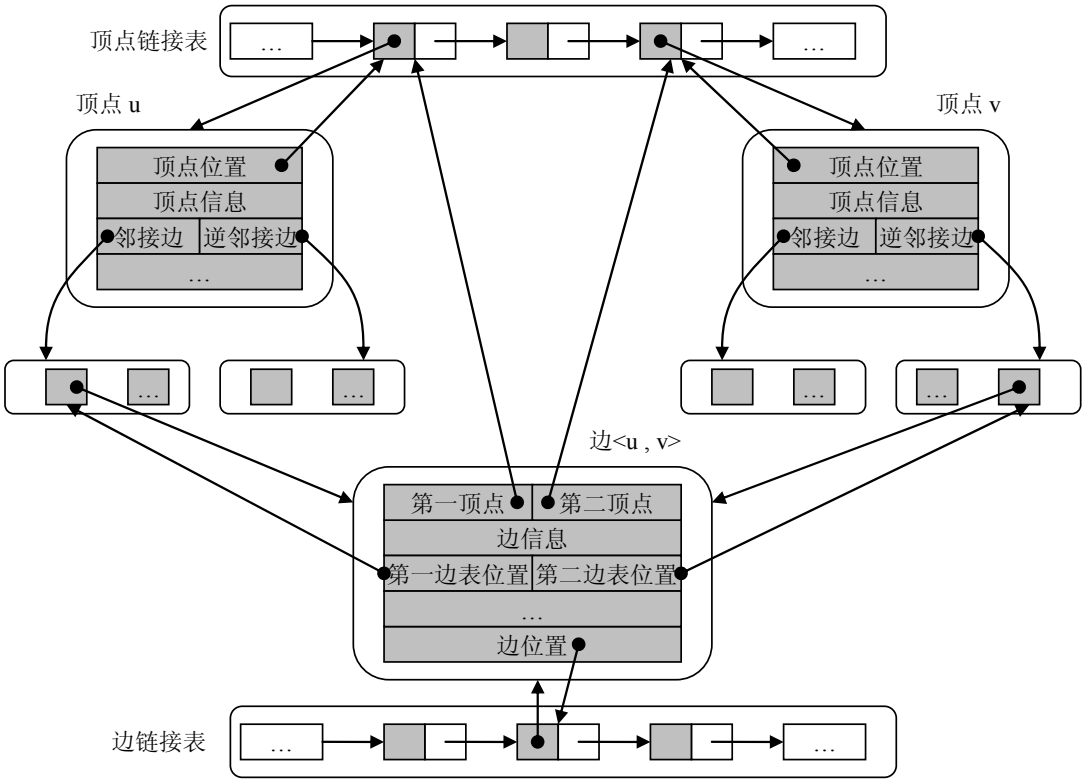


图 7-11 顶点与边的结构

在边中有 5 个重要的指针域：第一顶点域、第二顶点域、第一边表位置域、第二边表位置域、边位置域。在有向图中，第一顶点域指向该边的起始顶点在顶点表中的位置，第二顶点域指向该边的终止顶点在顶点表中的位置；如果是无向图，则分别指向边的两个顶点在顶点表中的位置；通过这两个域可以在 O(1)时间内定位与边关联的顶点。在有向图中，第一边表位置域指向边在其起始点的出边表中的位置，第二边表位置域指向边在其终止点的入边表中的位置；如果是无向图，则这两个域分别指向边在其第一、第二顶点的邻接边表（无向图的顶点只有邻接边表，无逆邻接边表）中的位置。边位置域指向边在边表中的位置，通过该域可以在 O(1)时间内定位边在图中的位置。

如此，存储了图中所有的顶点与边，以及顶点与边的相邻关系，则存储了整个图结构。

下面给出双链式存储结构中顶点与边的 Java 定义。

代码 7-3 双链式存储结构的顶点定义

```
public class Vertex {
    private Object info;        //顶点信息
    private LinkedList adjacentEdges;        //顶点的邻接边表
    private LinkedList reAdjacentEdges;    //顶点的逆邻接边表，无向图时为空
    private boolean visited;        //访问状态
```

```
private Node vexPosition;     //顶点在顶点表中的位置
private int graphType;        //顶点所在图的类型
private Object application;   //应用。如求最短路径时为 Path，求关键路径时为 Vtime
//构造方法:在图 G 中引入一个新顶点
public Vertex(Graph g, Object info) {
      this.info = info;
      adjacentEdges = new LinkedListDLNode();
      reAdjacentEdges = new LinkedListDLNode();
      visited = false;
      graphType = g.getType();
      vexPosition = g.insert(this);
      application = null;
}
//辅助方法:判断顶点所在图的类型
private boolean isUnDiGraphNode(){ return graphType==Graph.UndirectedGraph;}

//获取或设置顶点信息
public Object getInfo(){ return info;}
public void setInfo(Object obj){ this.info = info;}

//与顶点的度相关的方法
public int getDeg(){
      if (isUnDiGraphNode())
            return adjacentEdges.getSize();     //无向图顶点的(出/入)度为邻接边表规模
      else
            return getOutDeg()+getInDeg();    //有向图顶点的度为出度与入度之和
}
public int getOutDeg(){
      return adjacentEdges.getSize();           //有(无)向图顶点的出度为邻接表规模
}
public int getInDeg(){
      if (isUnDiGraphNode())
            return adjacentEdges.getSize();     //无向图顶点的入度就是它的度
      else
            return reAdjacentEdges.getSize(); //有向图顶点入度为逆邻接表的规模
}

//获取与顶点关联的边
public LinkedList getAdjacentEdges(){ return adjacentEdges;}
public LinkedList getReAdjacentEdges(){
      if (isUnDiGraphNode())
            return adjacentEdges;     //无向图顶点的逆邻接边表就是其邻接边表
      else
            return reAdjacentEdges;
```

```
    }

    //取顶点在所属图顶点集中的位置
    public Node getVexPosition(){ return vexPosition;}

    //与顶点访问状态相关方法
    public boolean isVisited(){ return visited;}
    public void setToVisited(){ visited = true;}
    public void setToUnvisited(){ visited = false;}

    //取或设置顶点应用信息
    protected Object getAppObj(){ return application;}
    protected void setAppObj(Object app){ application = app;}

    //重置顶点状态信息
    public void resetStatus(){
        visited = false;
        application = null;
    }
}
```

代码 7-3 说明：在 Vertex 中除了用于表示前面介绍的顶点中 3 个重要指针域的成员变量之外，还有 info、visited、graphType 和 application 四个成员变量。info 主要用于存储顶点的信息；visited 表示顶点的访问状态，在图的遍历、求最短路径等操作中使用；graphType 用来表示顶点所在图的类型，在有向图和无向图中顶点的操作实现有一些差别；application 也是在求最短路径等操作的实现中使用，具体的用法在后面详细介绍。

Vertex 中方法的基本功能如表 7-1 所示，而且方法的正确性不难理解。代码 7-3 中所有方法的时间复杂度均为 O(1)。

表 7-1 Vertex 类中方法的功能

序号	方法	功能描述
(1)	getInfo()	取顶点信息。
(2)	setInfo(info)	设置顶点信息。
(3)	getDeg()	返回点的度。
(4)	getOutDeg()	返回点的出度。
(5)	getInDeg()	返回点的入度。
(6)	getAdjacentEdges()	返回顶点的所有邻接边。
(7)	getReAdjacentEdges()	返回顶点的所有逆邻接边。
(8)	getVexPosition()	返回顶点在图顶点集中的位置，即在顶点链接表中的位置。
(9)	isVisited()	判断顶点在某操作实现中是否被访问过。
(10)	setToVisited()	将顶点访问状态设置为“已访问”。
(11)	setToUnvisited()	将顶点访问状态设置为“未访问”。
(12)	getAppObj()	取顶点应用状态信息。
(13)	setAppObj(obj)	设置顶点应用状态信息。
(14)	resetStatus()	重置顶点的所有状态信息，包括访问、应用状态。

在双链式存储结构中除了需要定义顶点还需要定义边，代码 7-4 给出了边的定义。

代码 7-4 双链式存储结构的边定义

```
public class Edge {
    public static final int NORMAL = 0;
    public static final int MST = 1;      //MST 边
    public static final int CRITICAL = 2;//关键路径中的边
    private int weight;                   //权值
    private Object info;                  //边的信息
    private Node edgePosition;            //边在边表中的位置
    private Node firstVexPosition;        //边的第一顶点与第二顶点
    private Node secondVexPosition;   //在顶点表中的位置
    private Node edgeFirstPosition;    //边在第一(二)顶点的邻接(逆邻接)边表中的位置
    private Node egdeSecondPosition;//在无向图中就是在两个顶点的邻接边表中的位置
    private int type;             //边的类型
    private int graphType;    //所在图的类型
    //构造方法:在图 G 中引入一条新边,其顶点为 u、v
    public Edge(Graph g, Vertex u, Vertex v, Object info){
        this(g,u,v,info,1);
    }
    public Edge(Graph g, Vertex u, Vertex v, Object info, int weight) {
        this.info = info;
        this.weight = weight;
        edgePosition = g.insert(this);
        firstVexPosition = u.getVexPosition();
        secondVexPosition = v.getVexPosition();
        type = Edge.NORMAL;
        graphType = g.getType();
        if (graphType==Graph.UndirectedGraph){
            //如果是无向图,边应当加入其两个顶点的邻接边表
            edgeFirstPosition = u.getAdjacentEdges().insertLast(this);
            egdeSecondPosition = v.getAdjacentEdges().insertLast(this);
        }else {
            //如果是有向图,边加入起始点的邻接边表,终止点的逆邻接边表
            edgeFirstPosition = u.getAdjacentEdges().insertLast(this);
            egdeSecondPosition = v.getReAdjacentEdges().insertLast(this);
        }
    }

    //get&set methods
    public Object getInfo(){ return info;}
    public void setInfo(Object obj){ this.info = info;}
    public int getWeight(){ return weight;}
    public void setWeight(int weight){ this.weight = weight;}
    public Vertex getFirstVex(){ return (Vertex)firstVexPosition.getData();}
```

```
    public Vertex getSecondVex(){ return (Vertex)secondVexPosition.getData();}
    public Node getFirstVexPosition(){ return firstVexPosition;}
    public Node getSecondVexPosition(){ return secondVexPosition;}
    public Node getEdgeFirstPosition(){ return edgeFirstPosition;}
    public Node getEdgeSecondPosition(){ return egdeSecondPosition;}
    public Node getEdgePosition(){ return edgePosition;}

    //与边的类型相关的方法
    public void setToMST(){ type = Edge.MST;}
    public void setToCritical(){ type = Edge.CRITICAL;}
    public void resetType(){ type = Edge.NORMAL;}
    public boolean isMSTEdge(){ return type==Edge.MST;}
    public boolean isCritical(){ return type==Edge.CRITICAL;}
}
```

代码 7-4 说明：在 Edge 中除了用于表示前面介绍的边中 5 个重要指针域的成员变量之外，还有 4 个成员变量：weight、info、graphType 、type。其中 info 和 weight 都是用来表示边的信息，但由于权值和边的其他信息相比更重要，并且会经常用到，因此将权值单独作为一个成员变量；graphType 与顶点中该变量的意义相同，是表示边所在图的类型；type 是用来表示边的类型的，目前只定义了 2 种特殊类型的边，一种是无向图最小生成树的边，一种是有向无环图关键路径中的边。Edge 类的构造方法是在图 G 中引入一条新边，因此边在加入边链接表的同时，还需要视图的类型将边加入与之关联的两个顶点的邻接边表或逆邻接边表中去；而且，在构造边时可以同时指定其权值，如果不指定则权值默认为 1。

Edge 中方法的基本功能如表 7-2 所示，而且方法的正确性不难理解。代码 7-4 中所有方法的时间复杂度均为 O(1)。

表 7-2 Edge 类中方法的功能

序号	方法	功能描述
(1)	getInfo()	取边信息。
(2)	setInfo(info)	设置边信息。
(3)	getWeight()	取边的权值。
(4)	setWeight(weight)	设置边的权值。
(5)	getFirstVex()	返回边的第一个顶点，有向图时的起始点。
(6)	getSecondVex()	返回边的第二个顶点，有向图时的终止点。
(7)	getFirstVexPosition()	返回边的第一个顶点在顶点集中的位置。
(8)	getSecondVexPosition()	返回边的第二个顶点在顶点集中的位置。
(9)	getEdgeFirstPosition()	返回边在第一顶点的边表中的位置。
(10)	getEdgeSecondPosition()	返回边在第二顶点的边表中的位置。
(11)	getEdgePosition()	返回边在图的边集中的位置。
(12)	setToMST()	将边设置为最小生成树中的边。
(13)	setToCritical()	将边设置为关键路径中的边。
(14)	resetType()	重置边的类型，设置为普通边。
(15)	isMSTEdge()	判断边是否是最小生成树中的边。
(16)	isCritical()	判断边是否是关键路径中的。

4.6 图 ADT 实现设计

与线性结构、树结构抽象数据类型实现不同，图结构的抽象数据类型实现不是简单的写一个类实现 Graph 接口即可。由于图有无向图和有向图之分，在 Graph 接口中有些接口方法是有向图支持的，有些是无向图支持的，有些是二者都支持的；并且在二者都支持的操作中有些操作的实现算法是一致的，有些操作实现的算法是有区别的；因此我们需要详细设计图 ADT 的实现方法。

一种简单的实现方法是编写两个类，一个类对应于有向图，一个类对应于无向图，这两个类分别实现 Graph 接口。然而这种实现会造成两个类中具有许多重复的代码（两个类都支持且具有相同算法的操作的代码），这样做既不利于代码的维护与管理，也违反了重构原则。为此，我们对图 ADT 的实现作如下设计：首先，确定无向图与有向图都支持的操作中实现算法相同的操作（见表 7-3），将这些操作的实现放在一个抽象类 AbstractGraph 中；其次，将两类图都支持但是实现算法不同的操作（见表 7-4）放在两个不同的类 DirectGraph 和 UndirectedGraph 中分别实现，当然 DirectGraph 和 UndirectedGraph 类都继承自 AbstractGraph 抽象类；最后，在 DirectGraph 类中实现只有有向图才支持的操作，在 UndirectedGraph 类中实现只有无向图才支持的操作。Graph 接口、AbstractGraph 抽象类、DirectGraph 类、UndirectedGraph 类之间的关系如图 7-12 所示。

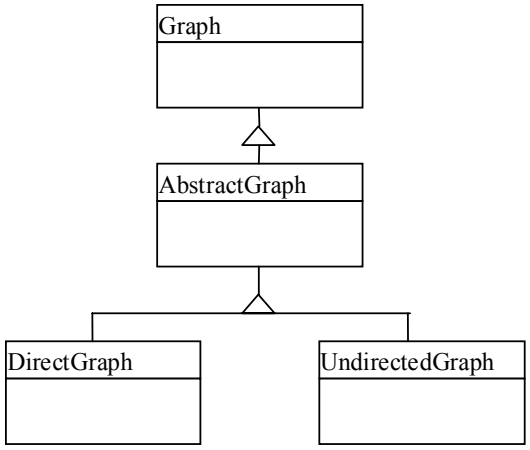


图 7-12 图 ADT 实现结构

上面介绍了图 ADT 实现中需要的类及其相互之间的关系，下面我们需要确定在各个类中实现的具体操作有哪些。可以在 AbstractGraph 抽象类中实现的操作由表 7-3 列出。

表 7-3 AbstractGraph 抽象类实现的方法

序号	方法	功能描述
(1)	getType()	返回当前图的类型。
(2)	getVexNum() getEdgeNum()	返回图中顶点数。返回图中边数。
(3)	getVertex() getEdge()	返回图中所有顶点的迭代器。返回图中所有边的迭代器。
(5)	insert(v) insert(e)	在图的顶点集中添加一个新顶点。在图的边集中添加一条新边。
(6)	areAdjacent(u, v)	判断顶点 v 是否为顶点 u 的邻接顶点。
(9)	DFSTraverse(v)	从顶点 v 开始深度优先搜索遍历图。
(10)	BFSTraverse(v)	从顶点 v 开始广度优先搜索遍历图。
(11)	shortestPath(v)	求顶点 v 到图中所有顶点的最短路径。

两类图都支持但是实现算法不同需要在 DirectGraph 类和 UndirectedGraph 类中分别实现的操作由表 7-4 列出。除此之外，还剩下 3 个操作 generateMST()、toplogicalSort()、criticalPath()。其中操作 generateMST()由无向图单独实现，操作 toplogicalSort()和 criticalPath()由有向图单独实现。在 DirectGraph 类和 UndirectedGraph 类中如果遇到不支持的操作则直接抛出代码 7-2 中定义的 UnsupportedOperation 异常。

表 7-4 DirectGraph 和 UndirectedGraph 分别实现的方法

序号	方法	功能描述
(4)	remove(v) remove(e)	在图中删除特定的顶点 v。在图中删除特定的边。
(7)	edgeFromTo(u, v)	返回从顶点 u 到顶点 v 的边，如果不存在返回空。
(8)	adjVertexs(u)	返回顶点 u 的所有邻接点。

图 ADT 的具体实现见本书提供的源代码，下面我们就图 ADT 所支持的操作中较为重要，并较为复杂的操作分为四节进行详细介绍。

4.7 图的遍历

和树的遍历类似，在图中也存在遍历问题。图的遍历就是从图中某个顶点出发，按某种方法对图中所有顶点访问且仅访问一次。图的遍历算法是求解图的连通性问题、拓扑排序和求关键路径等算法的基础。

图的遍历要比树的遍历要复杂的多。由于图中顶点关系是任意的，任一顶点都可能和其余的顶点相邻接；图可能是连通图也可能是非连通图；图中可能还存在环路，在访问了某个顶点之后，可能沿着某条搜索路径又回到该顶点。为了保证图中的各个顶点在遍历过程中访问且仅被访问一次，需要为每个顶点设一个访问标志，Vertex 类中的 visited 成员变量可以用来作为是否被访问过的标志。

对于图的遍历，通常有两种方法，即深度优先搜索和广度优先搜索。这两种遍历方法对有向图和无向图均适用，因此这两个操作在 AbstractGraph 抽象类中实现。

4.7.1 深度优先搜索

深度优先搜索（depth first search）遍历类似于树的先根遍历，是树的先根遍历的推广。

深度优先搜索的基本方法是：从图中某个顶点发 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

以图 7-13（a）中无向图为例，对其进行深度优先搜索遍历的过程如图 7-13（c）所示，其中黑色的实心箭头代表访问方向，空心箭头代表回溯方向，箭头旁的数字代表搜索顺序，顶点 a 是起点。遍历过程如下：首先访问顶点 a，然后

a) 顶点 a 的未曾访问的邻接点有 b、d、e，选择邻接点 b 进行访问；
b) 顶点 b 的未曾访问的邻接点有 c、e，选择邻接点 c 进行访问；
c) 顶点 c 的未曾访问的邻接点有 e、f，选择邻接点 e 进行访问；
d) 顶点 e 的未曾访问的邻接点只有 f，访问 f；
e) 顶点 f 无未曾访问的邻接点，回溯至 e；
f) 顶点 e 无未曾访问的邻接点，回溯至 c；
g) 顶点 c 无未曾访问的邻接点，回溯至 b；
h) 顶点 b 无未曾访问的邻接点，回溯至 a；
i) 顶点 a 还有未曾访问的邻接点 d，访问 d；
j) 顶点 d 无未曾访问的邻接点，回溯至 a。

到此，a 再没有未曾访问的邻接点，也不能向前回溯，从 a 出发能够访问的顶点均已访问，

并且此时图中再没有未曾访问的顶点，因此遍历结束。由以上过程得到的遍历序列为：a , b , c , e , f , d。

对于有向图而言，深度优先搜索的执行过程一样，例如图 7-13（b）中有向图的深度优先搜索过程如图 7-13（d）所示。在这里需要注意的是从顶点 a 出发深度优先搜索只能访问到 a , b , c , e , f，而无法访问到图中所有顶点，所以搜索需要从图中另一个未曾访问的顶点 d 开始进行新的搜索，即图 7-13（d）中的第 9 步。

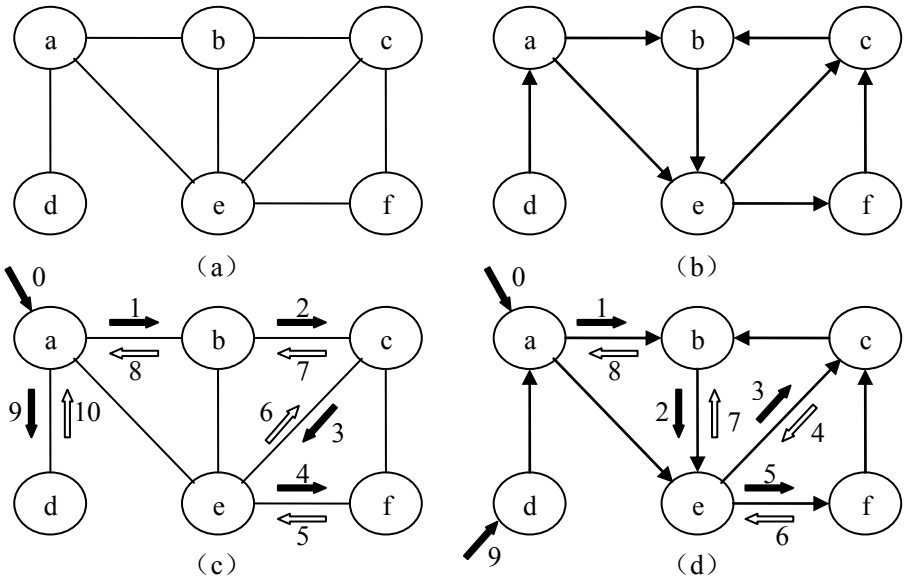


图 7-13 深度优先搜索

显然从某个顶点 v 出发的深度优先搜索过程是一个递归的搜索过程，因此可以简单的使用递归算法实现从顶点 v 开始的深度优先搜索。然而从 v 出发深度优先搜索未必能访问到图中所有顶点，因此还需找到图中下一个未曾访问的顶点，从该顶点开始重新进搜索。深度优先搜索算法的具体实现见算法 7-1。

算法 7-1 DFSTraverse

输入：顶点 v

输出：图深度优先遍历结果

代码：

```
public Iterator DFSTraverse(Vertex v) {
    LinkedList traverseSeq = new LinkedListDLNode();//遍历结果
    resetVexStatus();                    //重置顶点状态
    DFSRecursion (v, traverseSeq);    //从 v 点出发深度优先搜索
    Iterator it = getVertex(); //从图未曾访问的其他顶点重新搜索（调用图操作③）
    for(it.first(); !it.isDone(); it.next()){
        Vertex u = (Vertex)it.currentItem();
        if (!u.isVisited()) DFSRecursion(u, traverseSeq);
    }
    return traverseSeq.elements();
}
//从顶点 v 出发深度优先搜索的递归算法
private void DFSRecursion(Vertex v, LinkedList list){
    v.setToVisited();                //设置顶点 v 为已访问
```

```
        list.insertLast(v);                 //访问顶点 v
        Iterator it = adjVertexs(v);        //取得顶点 v 的所有邻接点（调用图操作⑧）
        for(it.first(); !it.isDone(); it.next()){
            Vertex u = (Vertex)it.currentItem();
            if (!u.isVisited()) DFSRecursion(u,list);
        }
    }
```

在算法 7-1 中对图进行深度优先搜索遍历时，对图中每个顶点最多调用一次 DFSRecursion 方法，因为一旦某个顶点已被访问，就不用再从该顶点出发进行搜索。因此，遍历图的过程实际就是查找每个顶点的邻接点的过程。当图采用双链式存储结构时，查找所有顶点的邻接点所需时间为 O(|E|)；除此之外，初始化顶点状态、判断每个顶点是否访问过以及访问图中所有顶点一次需要 O(|V|)时间。由此，当以双链式结构作为图的存储结构时，深度优先搜索遍历图的时间复杂度为 O(|V|+|E|)。

图的深度优先搜索算法也可以使用堆栈以非递归的形式实现，使用堆栈实现深度优先搜索的思想如下：

⑴ 首先将初始顶点 v 入栈；

⑵ 当堆栈不为空时，重复以下处理

栈顶元素出栈，若未访问，

则访问之并设置访问标志，将其未曾访问的邻接点入栈；

⑶ 如果图中还有未曾访问的邻接点，选择一个重复以上过程。

算法前两步的具体实现见算法 7-2，第三步与算法 7-1 中 DFSTraverse 方法实现类似，仅需要将从某个顶点 v 出发开始深度优先搜索的调用由原来的 DFSRecursion 改为调用 DFS。

算法 7-2 DFS

输入：顶点 v，链接表 list

输出：从顶点 v 出发的深度优先搜索

代码：

```
    //从顶点 v 出发深度优先搜索的非递归算法
    private void DFS(Vertex v, LinkedList list){
        Stack s = new StackSLinked();
        s.push(v);
        while (!s.isEmpty()){
            Vertex u = (Vertex)s.pop();          //取栈顶元素
            if (!u.isVisited()){                 //如果没有访问过
                u.setToVisited();                //访问之
                list.insertLast(u);
                Iterator it = adjVertexs(u);     //未访问的邻接点入栈（调用图操作⑧）
                for(it.first(); !it.isDone(); it.next()){
                    Vertex adj = (Vertex)it.currentItem();
                    if (!adj.isVisited()) s.push(adj);
                }//for
            }//if
        }//while
    }
```

4.7.2 广度优先搜索

广度优先搜索（breadth first search） 遍历类似于树的层次遍历，它是树的按层遍历的推广。

假设从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”先被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

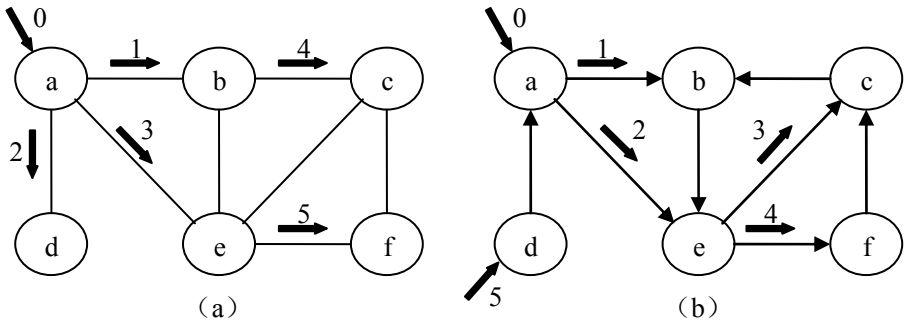


图 7-14 广度优先搜索

图 7-14（a）、（b）分别显示了对图 7-13（a）、（b）中两个图的广度优先搜索过程。对图 7-13（a）中无向图的广度优先搜索遍历序列为：a , b , d , e , c , f；对图 7-13（b）中有向图的广度优先搜索遍历序列为：a , b , e , c , f , d。同样，在这里从顶点 a 出发广度优先搜索只能访问到 a , b , e , c , f，所以搜索需要从图中另一个未曾访问的顶点 d 开始进行新的搜索，即图 7-14（b）中的第 5 步。

通过上述搜索过程，我们发现，广度优先搜索遍历图的过程实际上就是以起始点 v 为起点，由近至远，依次访问从 v 出发可达并且路径长度为 1、2、…的顶点。

广度优先搜索遍历的实现与树的按层遍历实现一样都需要使用队列，使用队列实现广度优先搜索的思想如下：

① 首先访问初始顶点 v 并入队；

② 当队列不为空时，重复以下处理

队首元素出队，访问其所有未曾访问的邻接点，并它们入队；

③ 如果图中还有未曾访问的邻接点，选择一个重复以上过程。

算法的具体实现见算法 7-3。

算法 7-3 BFSTraverse

输入：顶点 v

输出：图广度优先遍历结果

代码：

```
public Iterator BFSTraverse(Vertex v) {
    LinkedList traverseSeq = new LinkedListDLNode();//遍历结果
    resetVexStatus();              //重置顶点状态
    BFS(v, traverseSeq);           //从 v 点出发广度优先搜索
    Iterator it = getVertex();     //从图中未访问的顶点重新搜索（调用图操作③）
    for(it.first(); !it.isDone(); it.next()){
        Vertex u = (Vertex)it.currentItem();
```

```
            if (!u.isVisited()) BFS(u, traverseSeq);
        }
        return traverseSeq.elements();
    }
    private void BFS(Vertex v, LinkedList list){
        Queue q = new QueueSLinked();
        v.setToVisited();                   //访问顶点 v
        list.insertLast(v);
        q.enqueue(v);                       //顶点 v 入队
        while (!q.isEmpty()){
            Vertex u = (Vertex)q.dequeue();         //队首元素出队
            Iterator it = adjVertexs(u);            //访问其未曾访问的邻接点，并入队
            for(it.first(); !it.isDone(); it.next()){
                Vertex adj = (Vertex)it.currentItem();
                if (!adj.isVisited()){
                    adj.setToVisited();
                    list.insertLast(adj);
                    q.enqueue(adj);
                }//if
            }//for
        }//while
    }
```

在算法 7-3 中每个顶点最多入队、出队一次，遍历图的过程实际就是寻找队列中顶点的邻接点的过程，当图采用双链式存储结构时，查找所有顶点的邻接点所需时间为 O(|E|)，因此，算法 7-3 的时间复杂度为 O(|V|+|E|)。

4.8 图的连通性

4.8.1 无向图的连通分量和生成树

在对无向图进行遍历时，对于连通图，仅需从图中任何一个顶点出发，进行深度优先搜索或广度优先搜索，便可访问到图中所有顶点。对于非连通图，则需从多个顶点出发进行搜索，而每次从一个新的起始点出发进行搜索的过程中得到的顶点访问序列恰为其各个连通分量中的顶点集。

例如，图 7-15（a）中的图是非连通图，若从顶点 a 开始进行深度优先搜索遍历，在选择未曾访问的邻接点时按照顶点在图中的位置顺序（即 a , b , c , d , e , f , g , h）选择，2 次调用 DFS 方法（分别从 a、d 出发）得到的访问序列为

a , b , c , f , e　　d , g , h

这两个顶点集加上所有依附于它们的边，便构成了非连通图的 2 个连通分量如图图 7-15（b）所示。

设E是连通图G中所有边的集合，则从图中任意一个顶点出发遍历图时，必定将E分成两个子集：E_t和E_b，其中E_t是遍历图过程中经历的边的集合；E_b是剩余边的集合。显然E_t和图中所有顶点一起构成连通图G的极小连通子图，即图G的生成树。并且由深度优先搜索得到

的为深度优先搜索生成树；由广度优先搜索得到的为广度优先搜索生成树。

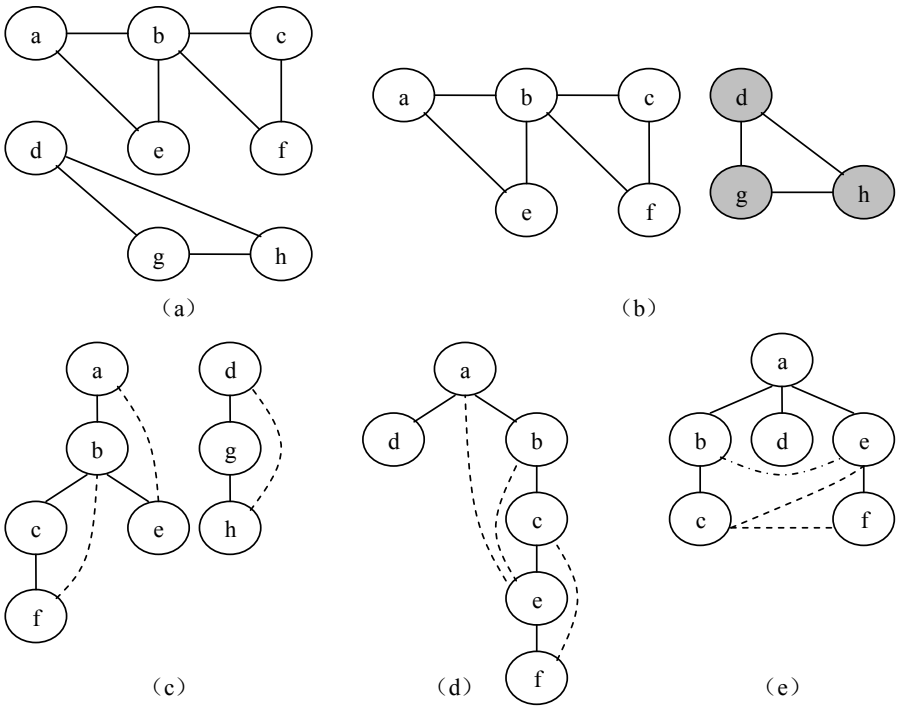


图 7-15 生成树与生成森林

例如图 7-15（d）和图 7-15（e）所示（不包括虚线代表的边）分别为图 7-13（a）中连通图的深度优先搜索生成树和广度优先搜索生成树，而图中虚线表示的边为集合E_b中的边。

对于非连通图，每个连通分量中的顶点集以及在遍历时走过的边一起构成若干棵生成树，这些连通分量的生成树组成非连通图的生成树森林。例如图 7-15（c）所示为图 7-15（a）的深度优先搜索森林，它由 2 棵深度优先搜索生成树组成。

4.8.2 有向图的强连通分量

在无向图中从某个顶点 v 出发深度优先搜索或广度优先搜索，就可以得到无向图中包含 v 在内的一个连通分量，然而从有向图中某个顶点 s 出发进行深度优先搜索或广度优先搜索，只能得到顶点 s 的可达分量，不一定能够得到包含 s 在内的强连通分量。

例如从图 7-13（b）中有向图顶点 a 出发进行深度优先搜索，可以访问到的顶点序列为：a , b , e , c , f，然而以这些顶点无法构成一个强连通分量，因为从 a 可以到达 b , e , c , f，但是从 b , e , c , f 却无法到达 a。下面我们来讨论在有向图中求强连通分量的算法。

对于任何有向边 e = <u , v>，称 R(e) = <v , u>为 e 的镜像边，即 R(e)的起点（终点）就是 e 的终点（起点）。对于任何有向图 G = (V , E)，我们称 R(E) = {R(e)|e∈E}为 E 的镜像边集，也就是说，集合 R(E)是由 E 中各边的镜像边组成。此时，也称 R(G) = (V , R(E))为 G 的镜像图。

为构造图 G = (V , E)中包含顶点 s 的强连通分量有如下方法：求出顶点 s 在图 G 中的可达分量与顶点 s 在 R(G)中的可达分量的交集；而在有向图中求顶点 s 的可达分量，只需要从 s 出发进行深度优先搜索或广度优先搜索即可。

例如在图 7-16（a）中灰色的顶点集为 a 在 G 中的可达分量；图 7-16（b）中灰色的顶点为 a 在 R(G)中的可达分量；图 7-16（c）中灰色的顶点为顶点 a 在图 G 中的可达分量与 a 在 R(G)中的可达分量的交集，它们以及与它们关联的边就构成了包含 a 的强连通分量。

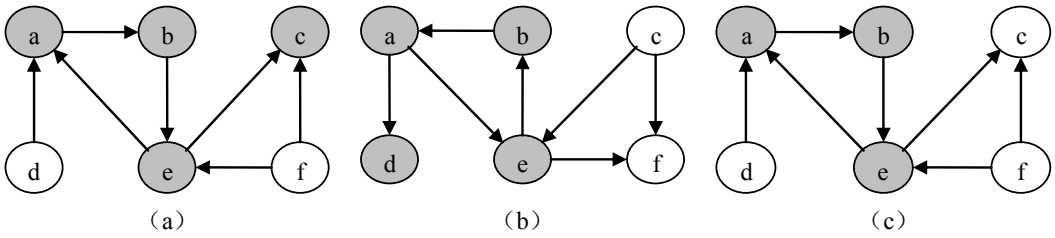


图 7-16 构造包含顶点 a 的强连通分量

如果需要确定有向图的所有强连通分量，可以从图中每个顶点出发重复上述操作，然而这种方法时间复杂度较高，实际上为确定有向图的所有强连通分量，只需要进行两次深度优先搜索即可，一次是在有向图 G 上进行，另一次是在 R(G)上进行。有兴趣的读者可以自行构造相应的算法。

4.8.3 最小生成树

通过上 7.5.1 小节的内容，我们看到对于连通图而言从图中不同顶点出发或从同一顶点出发按照不同的优先搜索过程可以得到不同的生成树。例如图 7-15（d）和图 7-15（e）所示就是同一个图的两棵不同生成树。

如此，对于一个连通网（连通带权图）来说，生成树不同，每棵树的代价（树中每条边上权值之和）也可能不同，我们把代价最小的生成树称为图的**最小生成树（minimum spanning tree）**。

最小生成树在许多领域都有重要的应用，例如利用最小生成树就可以解决如下工程中的实际问题：网络 G 表示 n 各城市之间的通信线路网线路，其中顶点表示城市，边表示两个城市之间的通信线路，边上的权值表示线路的长度或造价。可通过求该网络的最小生成树达到求解通信线路长度或总代价最小的最佳方案。

需要进一步指出的是，尽管最小生成树必然存在，但不一定唯一。

假设已知一个无向连通图 G = (V , E)，其边加权函数为 w: E→R，构造最小生成树的基本思想是：每步形成最小生成树的一条边；算法设置了边集合 A，初始时为空，该集合一直是某最小生成树的子集。在每步决定是否把边(u , v)添加到集合 A 中，其添加条件是 A∪{(u , v)}仍然是最小生成树的子集。我们称这样的边为 A 的安全边，因为可以安全地把它添加到 A 中而不会破坏上述条件。通过上述过程找到|V|-1 条边最后返回集合 A 时，A 就必然是一棵最小生成树。

上述构造最小生成树的思想中最关键的部分就是如何找到安全边(u , v)，下面我们将给出一条确认安全边的规则。

在介绍这一规则之前，我们先介绍几个概念。无向图G = (V, E)的一个**割**(S , V - S)是对顶点集的一个划分。图 7-16 说明了这个概念。当边(u , v)∈E的一个顶点在S中，而另外一个顶点在V-S中，我们说边(u , v)**横切**割(S , V - S)。在横切割的所有边中，权最小的边称为**轻边**。需要注意的是横切割的轻边可能不止一条。若集合A中没有边横切割，则我们说割不**妨害**边的集合A。例如图 7-17 中边(a , h)、(b , h)、(b , c)、(d , c)、(d , f)、(e , f)横切割(S , V - S)，其中(d , c)是轻边。子集A包含加粗的那些边，注意由于A中没有边横切割，所以割(S , V - S)不妨害A。

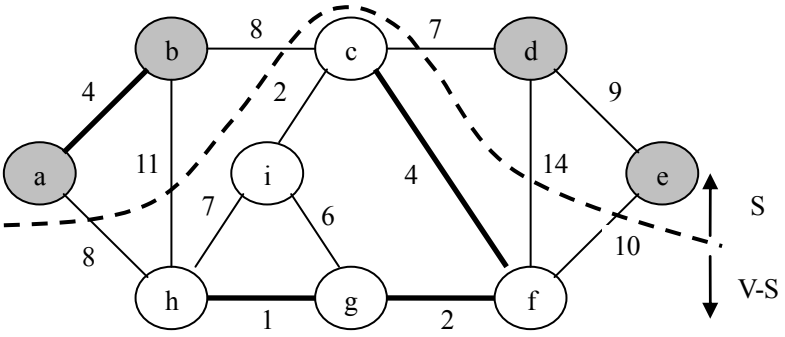


图 7-17 割及轻边的概念

定理 7.1 设图 G = (V, E)是一个无向连通图，且在 E 上定义了相应的加权函数 w，设 A 是 E 的一个子集且包含于 G 的某个最小生成树中，割(S , V - S)是 G 的不妨害 A 的任意割且边(u , v)是横切割(S , V - S)的一条轻边，则边(u , v)对集合 A 是安全的。

证明：

设 T'是包含 A 的一棵最小生成树。

如果 T'包含轻边(u , v)，则说明边(u , v)可以安全的加入 A 而不破坏 A 是某最小生成树的子集这一性质，因此边(u , v)对集合 A 是安全的。

如果T'不包含轻边(u , v)，由于T'是连通的，则在T'上必定存在一条不属于A的边(u' , v')横切割(S , V - S)，并且u和u'、v和v'之间均有路径相通。如此当将边(u , v)加入到T'中时，则在T'中产生一条包含(u , v)的回路，如果我们删除(u' , v')便可消除上述回路，同时得到另一棵生成树T。因为边(u , v)和(u' , v')横切割(S , V - S)，而边(u , v)是轻边，因此有边(u , v)的权值不大于(u' , v')的权值，即$w(u, v) \le w(u', v')$，而生成树T的代价

$$w(T) = w(T') - w(u', v') + w(u, v) \le w(T')$$

但T'是最小生成树，有$w(T') \le w(T)$，所以$w(T) = w(T')$，因此T必定也是最小生成树，如此边(u , v)对集合A是安全的。

证明完毕。

定理 7.1 使我们可以更好地了解前面算法思想在连通图G = (V , E)上的执行流程，算法开始时，A为空集，图G_A= (V , A)是一个森林，森林中包含|V|棵树，每个顶点对应一棵，一共|V|个连通分量。在算法执行过程中，边(u , v)是不妨害A的任意割的轻边，因此在A中加入这条安全的边(u , v) 都连接G_A中不同的连通分量，且不会在A中产生回路，并且使得A中边数加 1。每个迭代过程均将减少一棵树，当森林中只包含一棵树时，算法执行终止。

下述的两种最小生成树算法是对上述所介绍的算法思想的细化。在 Prim 算法中，集合 A 仅形成单棵树，加入集合 A 的安全边总是连结树与其他孤立顶点之间的轻边。在 Kruskal 算法中，集合 A 是一森林，加入集合 A 的安全边总是图中连结两不同连通分量的最小权边。

■ Prim 算法

假设G = (V, E)是连通网，A是G上最小生成树的边的集合。算法从S = {u_0}（$u_0 \in V$），A={}开始，重复执行下述操作：找到横切割(S , V - S) 的轻边(u_0 , v_0)并入集合A，同时v_0并入S，直到S = V为止。此时A中必定有|V|-1 条边，则T = (V , A)为G的最小生成树。

图 7-18 说明了 Prim 算法在图 7-17 中连通网上的执行过程。

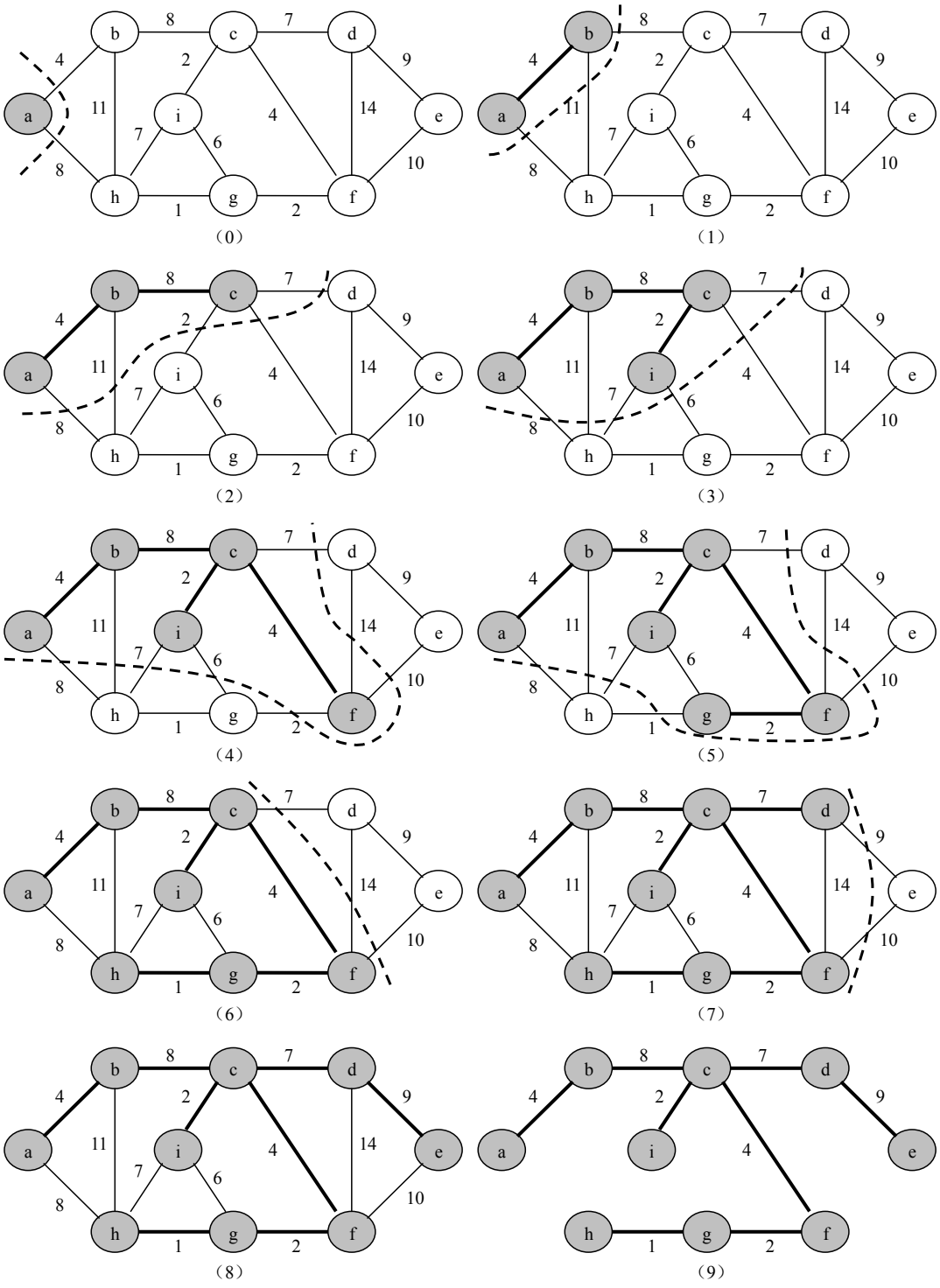


图 7-18 Prim 算法示例

在图 7-17 所示的构造最小生成树过程中，横切割(S , V - S) 的边会随着新加入 S 的顶点 k 变化而变化。为找到割(S , V - S) 的轻边，可以转化为如下操作：求出从 S 中顶点到达 V-S 中各个顶点的最短横切边，轻边是这些最短横切边中最小的一个。例如图 7-17（4）中，当 S = {a , b , c , i}时，到达 V-S 中每个顶点的最短横切边是：(c , d) = 7、(c , f) = 4、(i , g) = 6、(i , h) = 7、到 e 为∞，割(S , V - S) 的轻边为(c , f) = 4。

假设初始化时 S = {a}，从 S 到达 V-S 中各顶点的最短横切边初始化为 a 到其邻接顶点的距离即可，与 a 不相邻的设为∞。在算法执行过程中，会不断有新的顶点 k 加入 S，k 的

加入可能使得原本 V-S 中不可达的顶点变的可达或原本可达的顶点能以更小的代价可达，因此在 S 中引入新的顶点 k 后，需要以 k 为中间点更新到达 S-V 中各顶点的最短横切边。例如在图 7-17（3）中，当 S = {a , b , c}时，原本 g 不可达，到达 h 的最小距离为 8，当 i 加入 S 之后，g 变的可达并且到达 h 的最短横切边由(a , h) = 8 变为(i , h) = 7。

在算法的具体实现中，我们采用以下策略：首先，以顶点的成员变量 visited 来表示该顶点是否属于 S，visited = true 表示属于 S，否则不属于 S。其次，到达 V-S 中各个顶点的最短横切边通过该顶点的成员变量 application 来表示，此时 application 指向的是 Edge 类的对象，它是从 S 到达本顶点横切边中权值最小的一条。在构造最小生成树过程中，对顶点成员变量 application 的操作方法见代码 7-5。最后，最小生成树的表示采用设置图中边的类型来完成，即如果是最小生成树的边，将边的类型设置为 Edge.MST。

代码 7-5 求 MST 时，对 v.application 的操作

```
//获取到达顶点 v 的最小横切边权值
protected int getCrossWeight(Vertex v){
    if (getCrossEdge(v)!=null)
        return getCrossEdge(v).getWeight();
    else  return Integer.MAX_VALUE;
}
//获取到达顶点 v 的最小横切边
protected Edge getCrossEdge(Vertex v){ return (Edge)v.getAppObj();}
//设置到达顶点 v 的最小横切边
protected void setCrossEdge(Vertex v, Edge e){ v.setAppObj(e);}
```

Prim 算法的具体实现见算法 7-4。

算法 7-4 generateMST

输入：无向连通带权图

输出：构造最小生成树

代码：

```
public void generateMST(){
    resetVexStatus();    //重置图中各顶点的状态信息
    resetEdgeType();    //重置图中各边的类型信息
    Iterator it = getVertex();              //（调用图操作③）
    Vertex v = (Vertex)it.currentItem();//选第一个顶点作为起点
    v.setToVisited();                       //顶点 v 进入集合 S
    //初始化顶点集合 S 到 V-S 各顶点的最短横切边
    for(it.first(); !it.isDone(); it.next()){
        Vertex u = (Vertex)it.currentItem();
        Edge e = edgeFromTo(v,u);   //（调用图操作⑦）
        setCrossEdge(u,e);              //设置到达 V-S 中顶点 u 的最短横切边
    }
    for (int t=1;t<getVexNum();t++){        //进行|V|-1 次循环找到|V|-1 条边
        Vertex k = selectMinVertex(it);     //选择轻边在 V-S 中的顶点 k
        k.setToVisited();                   //顶点 k 加入 S
        Edge mst = getCrossEdge(k);         //割(S , V - S)  的轻边
        if (mst!=null) mst.setToMST();      //将边加入 MST
        //以 k 为中间顶点修改 S 到 V-S 中顶点的最短横切边
```

```
                Iterator adjIt = adjVertexs(k);        //取出 k 的所有邻接点
                for(adjIt.first(); !adjIt.isDone(); adjIt.next()){
                    Vertex adjV = (Vertex)adjIt.currentItem();
                    Edge e = edgeFromTo(k,adjV);            //（调用图操作⑦）
                    if (e.getWeight()<getCrossWeight(adjV))//发现到达 adjV 更短的横切边
                        setCrossEdge(adjV,e);
                }//for
            }//for(int t=1...
        }
        //查找轻边在 V-S 中的顶点
        protected Vertex selectMinVertex(Iterator it){
            Vertex min = null;
            for(it.first(); !it.isDone(); it.next()){
                Vertex v = (Vertex)it.currentItem();
                if(!v.isVisited()){ min = v; break;}
            }
            for(; !it.isDone(); it.next()){
                Vertex v = (Vertex)it.currentItem();
                if(!v.isVisited()&&getCrossWeight(v)<getCrossWeight(min))
                    min = v;
            }
            return min;
        }
```

算法 7-4 说明：算法的含义通过前面的分析不难理解。下面我们来分析算法的时间复杂度：首先，初始化部分所需时间为$O(|V|+|E|)$。其次，在|V|-1 次循环过程中，一方面要遍历各个顶点的邻接边，一共需要$O(|E|)$的时间；另一方面在每次循环中查找轻边需要对所有顶点遍历一遍，每次循环需时$O(|V|)$。因此算法的时间复杂度$T(n) = O(|V|+|E|) + O(|E|) + (|V|-1) O(|V|) = O(|V|^2+|E|) = O(|V|^2)$。

■ Kruskal 算法

Kruskal 算法的过程如下：假设 G = (V, E)是连通网，则令最小生成树的初始状态为只有|V|个顶点而无边的非连通图，图中每个顶点自成一个连通分量。在 E 中选择权值最小的边，若该边依附的顶点落在 T 中的不同连通分量上，则将此边加入 T，否则舍去此边选择下一条代价最小的边。以此类推，直到所有顶点都在同一连通分量上为止。

图 7-19 说明了 Kruskal 算法在图 7-17 中连通网上的执行过程。

Kruskal算法的时间复杂度主要取决于在E中选择权值最小的边，以及判断边是否落在两个连通分量上。选择E中权值最小的边，可以先对|E|条边排序，然后依次取出各边即可，运算需间$O(|E| \log |E|)$（排序的内容见第十章）。在算法的实现中可以使用不相交集[①]数据结构以保持数个互相分离的元素集合。每一集合包含当前森林中某个树的结点，操作FIND-SET(u)（查找）返回包含u的集合的一个代表元素，因此我们可以通过FIND-SET(v)来确定两结点u和v是否属于同一棵树，通过操作UNION（合并）来完成树与树的联结。在不相交集中使用按秩合并和通路压缩的方法来实现不相交集，由于从渐近复杂度上来说这是目前所知的最快

[①] 见参考文献[]

的实现方法。对于Kruskal算法有|V|-1 次合并和 2|E|次查找，运算的总花费为 O (|E| log *|V|)[①]。因此算法总的时间取决于排序步，即 O (|E| log |E|)。

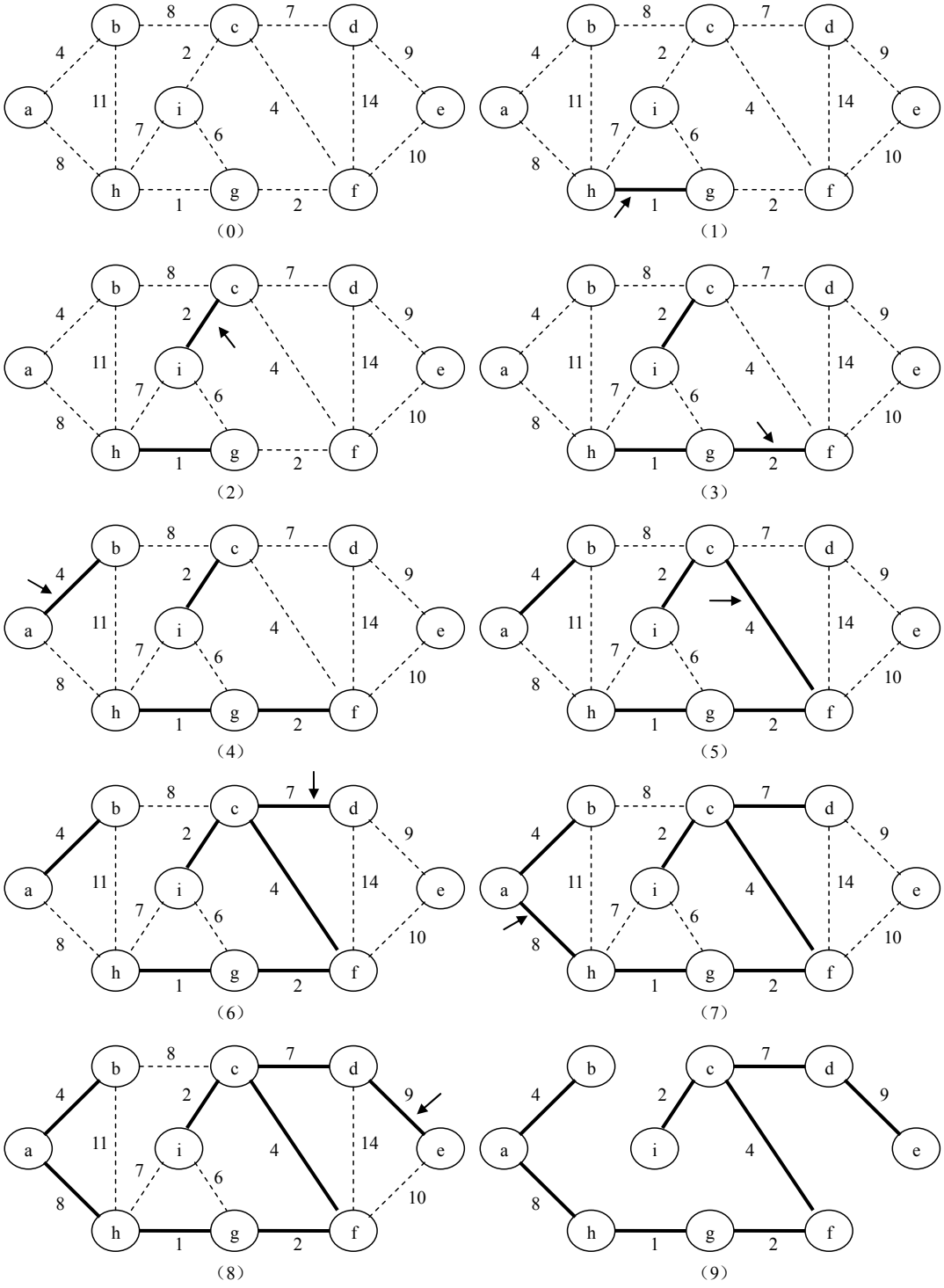


图 7-18 Kruskal 算法示例

① $$\log^* n = \begin{cases} 0 & n=0 \text{ 或 } 1 \\ \min\{i \geq 0 \mid \underbrace{\log \log \ldots \log}_{i \text{ 个}} n \} \leq 1 & n \geq 2 \end{cases}$$

4.9 最短距离

在许多应用领域，带权图都被用来描述某个网络，比如通信网络、交通网络等。这种情况下，各边的权重就对应于两点之间通信的成本或交通费用。此时，一类典型的问题就是：在任意指定的两点之间如果存在通路，那么最小的消耗是多少。这类问题实际上就是带权图中两点之间最短路径的问题。

在求解最短路径问题时，有时对应实际情况带权图应当是有向图，例如同一信道两个方向的信息流量不同，会造成信息从终端 A 到 B 和从终端 B 到 A 所需的时延不同；而有时对应于实际情况的带权图可以是无向图，例如从城市 A 到 B 和从城市 B 到 A 的公路长度都一样。下面我们将要介绍的 Dijkstra 算法和 Floyd 算法对于带权无向图或有向图都适用。

在图中两点之间的最短路径问题包括两个方面：一是求图中一个顶点到其他顶点的最短路径，二是求图中每对顶点之间的最短路径。

4.9.1 单源最短路径

单源最短路径是指，在带权图 G = (V, E)中，已知源点为 s∈V，求 s 到其余各顶点的最短路径。显然在图中若顶点 v 是从源点 s 可达的，那么从 s 到顶点 v 的最短路径必然存在，其长度称为“从 s 到 v 的最短距离”，记作 δ(s , v)。如果顶点 v 从 s 不可达，则可以认为从 s 到 v 的距离为∞。需要指出的是，两点之间的最短路径可能不是唯一的。

在介绍求解单源最短路径问题的算法之前先介绍最短路径的一条基本性质。

定理 7.2 若$\pi = (u_0=s , u_1 , u_2 , \dots , u_k=v)$是从顶点s到顶点v的最短路径，则对于任何$0\le i<j\le k$，$\tau = (u_i , u_{i+1} , \dots , u_j)$是从顶点$u_i$到$u_j$的最短路径。

定理 7.2 也可以简单的描述为：最短路径的子路径也是最短路径。

证明：

如图 7-19 所示，顶点u_i和u_j将从顶点s到顶点v的最短路径π分成 3 部分，分别记为$\rho = (s , u_1 , u_2 , \dots , u_i)$，$\tau= (u_i , u_{i+1} , \dots , u_j)$，$\sigma = (u_j , u_{i+1} , \dots , u_k)$。

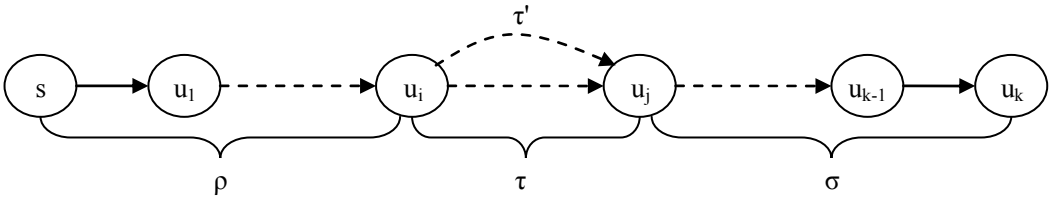


图 7-19 最短路径的性质

反证，假设$\tau= (u_i , u_{i+1} , \dots , u_j)$不是从顶点$u_i$到$u_j$的最短路径，那么必定存在另外一条路径是从顶点$u_i$到$u_j$的最短路径，不妨设为τ'，于是有$|\tau'| < |\tau|$。同时$\rho + \tau' + \sigma$也是一条从s到$u_k$的路径，并且其长度$|\rho + \tau' + \sigma| = |\rho| + |\tau'| + |\sigma| < |\rho| + |\tau| + |\sigma| = |\pi|$，这说明从s到$u_k$还有一条长度小于π的路径，即π不是从s到$u_k$的最短路径，与已知矛盾。

证明完毕。

如果考虑到在本章中边的权值为正数，则由定理 7.2 可以得到以下结论：若$\pi = (s , u_1 , u_2 , \dots , u_k)$是从s到$u_k$的最短路径，则从s到各顶点$u_i$(i=1, 2, …k)的最短路径是严格递增的。

为求解单源最短路径问题 Dijkstra 提出了一个算法，该算法是按照最短路径长度递增的顺序产生从源点 s 到其余|V|-1 个顶点的|V|-1 条最短路径的。

Dijkstra 算法将带权图 G = (V, E)的顶点分为两个集合：S 、V - S，其中

- 顶点集 S 是已求出的最短路径的终点集合（初始时 S = {s}）。
- 顶点集 V-S 是尚未求出最短路径的终点的集合。

算法将按最短路径长度递增的顺序逐个将 V-S 中的顶点加入到 S 中，直到所有顶点都被加入到 S 中为止。

算法为每个顶点 v 定义了一个变量 distance，该变量纪录了从 s 出发，经由 S 中的顶点到达 v 的当**前最短距离**。初始时，每个顶点的当前最短距离 distance 为图中从 s 到 v 的边的权值，如果从 s 到 v 没有边则 distance(v) = ∞，并且 distance(s) = 0。

假设在算法执行过程中的某个时刻，S = {s , u_1 , u_2 , … , u_{k-1} }，下一条最短路径的终点是u_k，那么下一条最短路径或者是由s直接到达，或者是只经过S中的某些顶点而后到达。这一结论可用反证法证明，假设下一条最短路径上第一个不属于S的顶点为y，从s到u_k的最短路径π = (s , … , y , … , u_k)，因为π是最短路径，根据定理 7.2 知ρ = (s , … , y)是从s到y的最短路径，且δ(s , y) < δ(s , u_k)。由于y∈V-S并且u_k∈V-S，那么按照我们选择顶点的原则应当先选择y，而不是u_k，这与下一条最短路径的终点是u_k矛盾。

根据以上结论，长度最短的一条最短路径必为π = (s , u_k)，其中u_k满足：

distance (u_k) = min{ distance (u_i)| u_i ∈V-S}

在求得顶点u_k的最短路径后，将u_k并入S，即S = S ∪{ u_k }。每当一个新的顶点u_k加入S之后，则对V-S中的顶点而言，多了一个从s到自身的“中转”顶点，从而可能出现从s出发途经u_k到达自身的新路径，这条路径可能比该顶点的当前最短路径更短，因此当S中加入新的顶点u_k后，要以u_k为“中转”顶点对V-S中各个顶点的当前最短路径distance进行修正。

修正V-S中各顶点的当前最短路径的方法是：依次对对顶点u_k的邻接点u_i（u_i∈V-S）执行如下操作：

distance (ui) = min{ distance (ui) , distance (uk)+w(uk , ui)}　　ui∈V-S

其中distance (u_i)是u_i当前最短距离，distance (u_k)+w(u_k , u_i)是经过u_k “中转”的路径长度，修正之后u_i的当前最短路径长度是两者中小的。

修正 V-S 中各个顶点的 distance 后，再选择 distance 最小的顶点加入 S，这一过程一直进行下去，直到 S = V。

整理以上算法思想，得到 Dijkstra 算法的基本执行过程如下：

① 初始化：

S = {s}；distance(s) = 0；

distance(u_i) = w(s , u_i)或∞，（u_i∈V-S）；

② 选择distance (u_k) = min{ distance (u_i)| u_i ∈V-S}，u_k为下一条最短路径的终点；

③ S = S ∪{ u_k }

④ 以u_k为“中转”，修正V-S中各个顶点distance：

distance (u_i) = min{ distance (u_i) , distance (u_k)+w(u_k , u_i)}　u_i∈V-S

⑤ 重复②—④步|V|-1 次。

图 7-20 图示了 Dijkstra 算法在图 7-20（a）的有向带权图上求顶点 a 到其他顶点最短路径的过程。

在算法的具体实现中，我们采用以下策略：首先，以顶点 v 的成员变量 visited 来表示该顶点是否属于 S，visited = true 表示属于 S，否则属于 V-S。其次，从 s 到达 V-S 中各个顶点 v 的最短路径通过 v 的成员变量 application 来表示，此时 application 指向的是 Path 类的对象。该对象是从 s 到达 v 的当前最短路径，其中包含 v 的当前最短距离 distance，以及取得最短距离的路径上途经的顶点。在 Dijkstra 算法执行过程中，对顶点成员变量 application 的操作方法见代码 7-6。最后，求得从 s 到其余顶点的所有最短路径通过迭代器对象返回。

代码 7-6　Dijkstra 算法中，对 v.application 的操作

```
    //取或设置顶点 v 的当前最短距离
    protected int getDistance(Vertex v){ return ((Path)v.getAppObj()).getDistance();}
    protected void setDistance(Vertex v, int dis){ ((Path)v.getAppObj()).setDistance(dis);}
    //取或设置顶点 v 的当前最短路径
    protected Path getPath(Vertex v){ return (Path)v.getAppObj();}
    protected void setPath(Vertex v, Path p){ v.setAppObj(p);}
```

其中 Path 类的定义见代码 7-7。

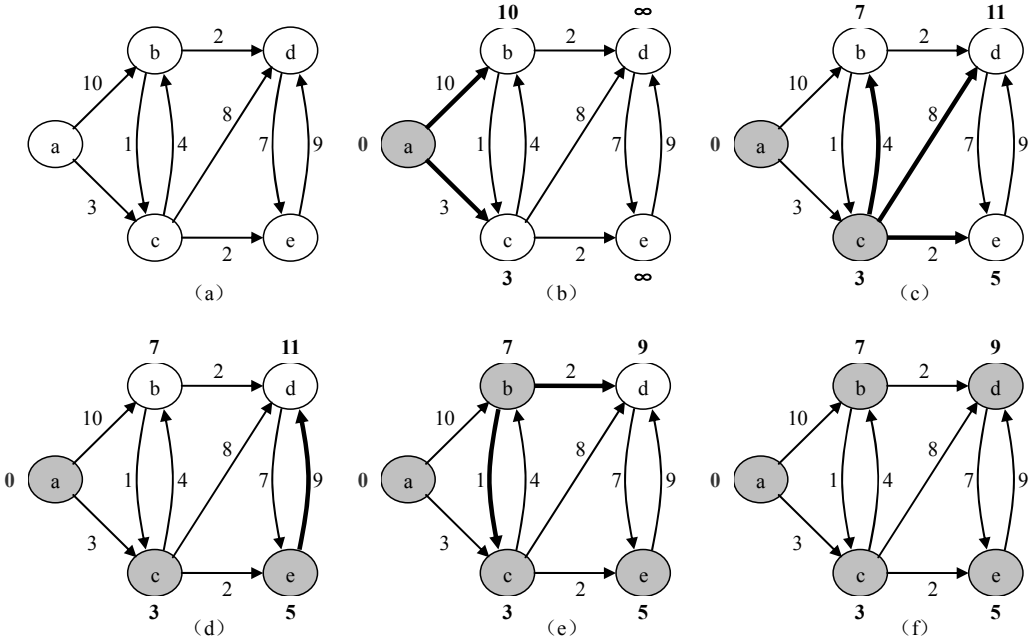


图 7-20 Dijkstra 算法执行过程

代码 7-7　Path 类定义

```
public class Path {
    private int distance;            //起点与终点的距离
    private Vertex start;            //起点信息
    private Vertex end;              //终点信息
    private LinkedList pathInfo;  //起点到终点途经的顶点序列
    //构造方法
    public Path() {    this(Integer.MAX_VALUE,null,null);    }
    public Path(int distance, Vertex start, Vertex end) {
        this.distance = distance;
        this.start = start;
        this.end = end;
        pathInfo = new LinkedListDLNode();
    }
    //判断起点与终点之间是否存在路径
    public boolean hasPath() {
        return distance!=Integer.MAX_VALUE&&start!=null&&end!=null;
    }
    //求路径长度
    public int pathLength(){
```

```
            if (!hasPath()) return -1;
            else if (start==end) return 0;
            else return pathInfo.getSize()+1;
        }
        //get&set methods
        public void setDistance(int dis){ distance = dis;}
        public void setStart(Vertex v){ start = v;}
        public void setEnd(Vertex v){ end = v;}
        public int getDistance(){ return distance;}
        public Vertex getStart(){ return start;}
        public Vertex getEnd(){ return end;}
        public Iterator getPathInfo(){
            return pathInfo.elements();
        }
        //清空路经信息
        public void clearPathInfo(){
            pathInfo = new LinkedListDLNode();
        }
        //添加路径信息
        public void addPathInfo(Object info){
            pathInfo.insertLast(info);
        }
}
```

算法 7-5 给出了 Dijkstra 算法的具体实现。

算法 7-5 shortestPath

输入：顶点 v

输出：v 到其他顶点的最短路径

代码：

```
    public Iterator shortestPath(Vertex v) {
        LinkedList sPath = new LinkedListDLNode();    //所有的最短路径序列
        resetVexStatus();                    //重置图中各顶点的状态信息
        //初始化，将 v 到各顶点的最短距离初始化为由 v 直接可达的距离
        Iterator it = getVertex();           //（调用图操作③）
        for(it.first(); !it.isDone(); it.next()){
            Vertex u = (Vertex)it.currentItem();
            int weight = Integer.MAX_VALUE;
            Edge e = edgeFromTo(v,u);  //（调用图操作⑦）
            if (e!=null)     weight = e.getWeight();
            if(u==v)         weight = 0;
            Path p = new Path(weight,v,u);
            setPath(u, p);
        }
        v.setToVisited();          //顶点 v 进入集合 S
        sPath.insertLast(getPath(v));       //求得的最短路径进入链接表
```

```
        for (int t=1;t<getVexNum();t++){  //进行|V|-1 次循环找到|V|-1 条最短路径
            Vertex k = selectMin(it);                    //找 V-S 中 distance 最小的点 k
            k.setToVisited();                            //顶点 k 加入 S
            sPath.insertLast(getPath(k));                //求得的最短路径进入链接表
            int distK = getDistance(k);                  //修正 V-S 中顶点当前最短路径
            Iterator adjIt = adjVertexs(k);              //取出 k 的所有邻接点
            for(adjIt.first(); !adjIt.isDone(); adjIt.next()){
                Vertex adjV = (Vertex)adjIt.currentItem();    //k 的邻接点 adjV
                Edge e = edgeFromTo(k,adjV);             //（调用图操作⑦）
                //发现更短的路径
                if ((long)distK+(long)e.getWeight()<(long)getDistance(adjV)){
                    setDistance(adjV, distK+e.getWeight());
                    amendPathInfo(k,adjV); //以 k 的路径信息修改 adjV 的路径信息
                }
            }//for
        }//for(int t=1...
        return sPath.elements();
    }
    //在顶点集合中选择路径距离最小的
    protected Vertex selectMin(Iterator it){
        Vertex min = null;
        for(it.first(); !it.isDone(); it.next()){
            Vertex v = (Vertex)it.currentItem();
            if(!v.isVisited()){ min = v; break;}
        }
        for(; !it.isDone(); it.next()){
            Vertex v = (Vertex)it.currentItem();
            if(!v.isVisited()&&getDistance(v)<getDistance(min))
                min = v;
        }
        return min;
    }
```

算法 7-5 说明：通过前面的分析算法的含义不难理解。下面我们来分析算法的时间复杂度。第一步初始化需要O(|V|)的时间，接着进入一个|V|-1 重的循环，在循环中selectMin(it)是对图中所有顶点进行遍历，以找到下一条最短路径，该方法需要O(|V|)时间。在循环中，还需要通过每次找到的“中转”顶点k来修正V-S中顶点的distance，由于该过程只需要对k的邻接边进行遍历，所以在所有循环过程中只需要O(|E|)时间。因此算法的时间复杂度为O(|V|) + (|V|-1)×O(|V|) + O(|E|) = O($|V|^2$ + |E|) = O($|V|^2$)。

4.9.2 任意顶点间的最短路径

Dijkstra算法只能求出源点到其余顶点的最短路径，如果需要求出带权图中任意一对顶点之间的最短路径，可以用每一个顶点作为源点，重复调用Dijkstra算法|V|次，时间复杂度为O($|V|^3$)。下面介绍一种形式更简洁的方法，即Floyd算法，其时间复杂度也是O($|V|^3$)。

假设求出的每对顶点之间的最短距离使用一个|V|×|V|矩阵D保存和输出。下面定义符号$D^{(k)}$，$0 \le k \le |V|$。在定义中假设带权图中所有的顶点排成一个序列。

定义：$D^{(k)}$（$0 \le k \le |V|$）是一个|V|阶方阵，其中$D^{(k)}[i][j]$是在考虑带权图中前k个顶点，将它们作为中间顶点时从顶点v_i到顶点v_j的当前最短距离（$1 \le i \le |V|$，$1 \le j \le |V|$）。

$D^{(0)}$表示当顶点v_i，v_j之间不考虑任何顶点作为中间顶点时的最短距离，显然$D^{(0)}[i][j]$就是顶点v_i到v_j的边的权值，如果使用邻接矩阵A作为存储结构，$D^{(0)}[i][j]$=A[i][j].weight。并且如果将所有的顶点均考虑在内，v_i到v_j的当前最短距离就是在图中v_i到v_j的最短距离，即$\delta(v_i, v_j) = D^{(|V|)}[i][j]$ （$1 \le i \le |V|$，$1 \le j \le |V|$）。

Floyd 算法的基本思想是：

(1) 用邻接矩阵初始化$D^{(0)}$，对角线元素为 0；

(2) 在顶点v_i、v_j之间考虑顶点v_1，比较在引入v_1之后v_i到v_j的当前最短距离是否可以通过v_1变得更小。把v_1放在v_i到v_j的路径上，v_i到v_j之间可能会产生新的路径，其距离为$D^{(0)}[i][1] + D^{(0)}[1][j]$，当然$v_1$的引入可能反而会加大$v_i$到$v_j$的距离，因此需要比较$D^{(0)}[i][1] + D^{(0)}[1][j]$与$D^{(0)}[i][j]$的大小。最终，在考虑$v_1$之后$v_i$到$v_j$的当前最短距离为两者中小的。即

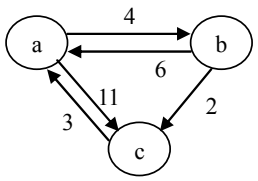
$$D^{(1)}[i][j] = \min\{ D^{(0)}[i][1] + D^{(0)}[1][j] , D^{(0)}[i][j]\}$$

(3) 一般情况下，如果在考虑了前k-1 个顶点$\{v_1 , v_2 , \dots , v_{k-1}\}$之后，从顶点$v_i$到$v_j$的当前最短距离是$D^{(k-1)}[i][j]$。那么在顶点$v_i$、$v_j$之间考虑前k个顶点时，顶点$v_i$到$v_j$的当前最短距离为以下两个距离中小的：在考虑前k-1 个顶点基础上将v_k放在v_i到v_j的路径上，此时产生新的路径长度为$D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$；以及不将$v_k$放在$v_i$到$v_j$的路径上的距离$D^{(k-1)}[i][j]$。最终，在考虑前k个顶点$\{v_1 , v_2 , \dots , v_k\}$之后，$v_i$到$v_j$的当前最短距离

$$D^{(k)}[i][j] = \min\{ D^{(k-1)}[i][k] + D^{(k-1)}[k][j] , D^{(k-1)}[i][j]\} \qquad 1 \le k \le |V|$$

(4) 依次进行下去，直到k = |V|。此时$D^{(|V|)}[i][j]$即为带权图中任意两个顶点v_i到v_j的最短距离。

图 7-21 图示了Floyd算法执行的过程。其中$D^{(k)}$定义如上，$P^{(k)}$表示当前最短路径。



D	D^0			D^1			D^2			D^3		
	a	b	c	a	b	c	a	b	c	a	b	C
a	0	4	11	0	4	11	0	4	**6**	0	4	6
b	6	0	2	6	0	2	6	0	2	**5**	0	2
c	3	∞	0	3	**7**	0	3	7	0	3	7	0
P	P^0			P^1			P^2			P^3		
	a	b	c	a	b	c	a	b	c	a	b	c
a		a,b	a,c		a,b	a,c		a,b	**a,b,c**		a,b	a,b,c
b	b,a		b,c	b,a		b,c	b,a		b,c	**b,c,a**		b,c
c	c,a			c,a	**c,a,b**		c,a	c,a,b		c,a	c,a,b	

图 7-20 Floyd 算法执行过程

Floyd算法的具体实现，读者可以在构造出邻接矩阵的基础上自己实现。Folyd算法共进行|V|次循环，每次循环处理|V|×|V|矩阵的每个元素，因此算法的时间复杂度为$O(|V|^3)$。

4.10 有向无环图及其应用

有向无环图（directed acyclic graph）是指一个无环的有向图，简称 DAG。有向无环图是描述一项工程或系统进行过程的有效工具。除最简单的情况之外，几乎所有的工程都可分为若干个称做**活动（activity）**的子工程，而这些子工程之间，通常受着一定条件的约束，如其中某些子工程的开始必须在另一些子工程完成之后。

对整个工程和系统，人们关心的是两个方面的问题：一是能否顺利进行，应该如何进行；二是估算整个工程完成所必须的最短时间，对应于有向图，即为进行拓扑排序和求关键路径的操作。

4.10.1拓扑排序

如果在一个有向图G = <V , E>中用顶点表示活动，用有向边<v_i , v_j>表示活动v_i必须先于活动v_j进行。这种有向图叫做顶点表示活动的**AOV网络（activity on vertices networks）**。

例如，一件商品的生产就是一项工程，它可以用一个 AOV 网络来表示，如图 7-21（a）所示。假设该商品的生产包含 4 项活动：

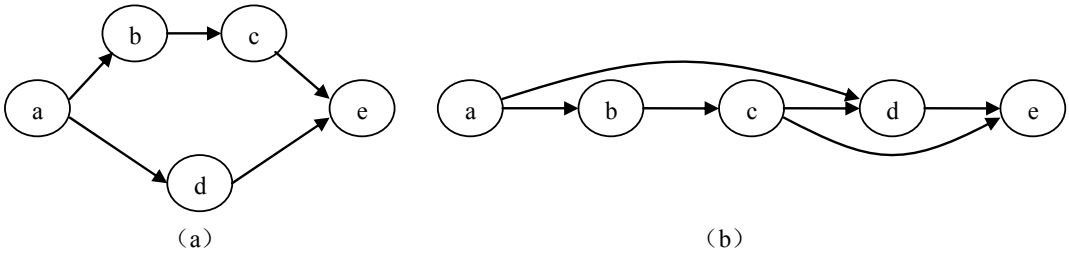
a.购买原材料；

b.生产零件 1；

c.用零件 1 加工零件 2；

d.生产零件 3；

e.组装零件 2、3 得到成品。



（a） （b）

图 7-21 AOV 网及拓扑序列

现在的问题是如何判断某个 AOV 网络所表示的工程是否可以完成；以及如何完成，即各个活动按照什么顺序完成。如果能够给出一个活动序列，该活动序列包含 AOV 网中所有顶点表示的活动，并且该活动序列满足 AOV 网络中所有应存在的前驱和后继关系。则该活动序列就是一种完成工程的方法。例如图 7-21（a）所示的工程可以图 7-21（b）所示的活动序列：a , b , c , d , e 来完成整个工程。

通过上面的例子可以看到，在一个AOV网络中，若v_i为v_j的先行活动，v_j为v_k的先行活动，则v_i必为v_k的先行活动，即活动的先行关系具有传递性。如果从离散数学的观点看AOV网络中的活动关系可以看成是一个偏序关系，而上述给出工程完成活动的线性序列可以看成是一个全序关系。

由某个集合上的一个偏序得到该集合上的一个全序，此操作称之为**拓扑排序（topological sort）**。即将 AOV 网络各个顶点 （代表各个活动）排列成一个线性有序的序列，使得 AOV 网络中所有应存在的前驱和后继关系都能得到满足。拓扑排序就是构造 AOV 网络顶点的拓扑有序序列的运算。

在一个 AOV 网络中是不能存在环的，因为如果存在环，说明某项活动的开始必须是以

自身的结束作为前提的。需要注意的是 AOV 网络的拓扑序列不是唯一的，例如图 7-21（a）的另一个拓扑序列为：a , d , b , c , e。

为得到 AOV 网络的拓扑序列，可以使用以下方法：

⑴ 在 AOV 网络中选一个没有直接前驱的顶点，并输出之；

⑵ 从图中删去该顶点，同时删去所有它发出的有向边；

⑶ 重复以上⑴、⑵步, 直到全部顶点均已输出，或图中不存在无前驱的顶点。

图 7-22 图示了拓扑排序的过程。

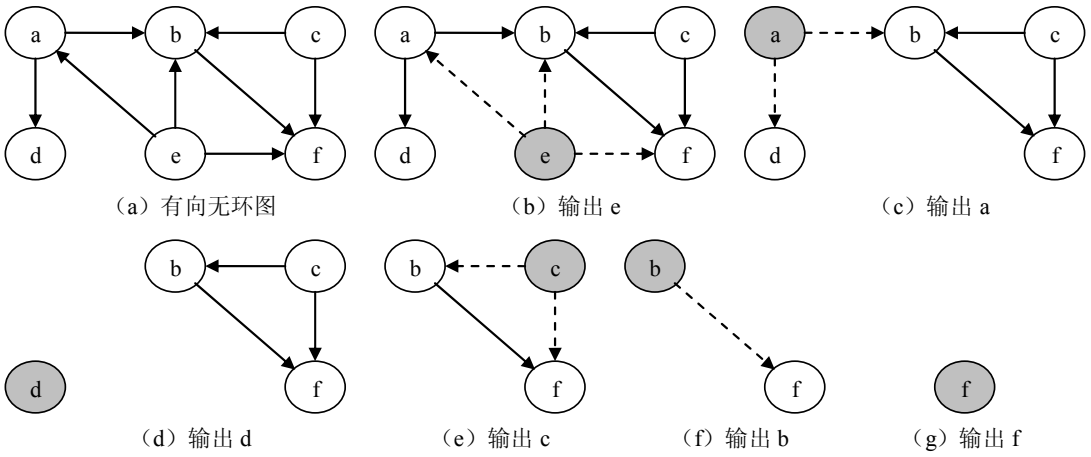


图 7-22 拓扑排序的过程
得到拓扑序列为：e , a , d , c , b , f

以上拓扑排序过程的正确性不难理解，但在计算机中如何实现？针对上述操作的两个关键步骤，我们作以下处理：首先，为每个顶点 v 设置一个变量，记录其在拓扑排序过程中的入度，入度为 0 的顶点就是没有直接前驱的顶点。其次，删除一条有向边可以用有向边终止点的入度减 1 来实现。由此，构造一个 AOV 网络的拓扑序列的算法可以描述如下：

⑴ 建立入度为零的顶点栈；

⑵ 当入度为零的顶点栈不空时，重复执行

从顶点栈中退出一个顶点，并输出之；

搜索以这个顶点发出的边，将边的终顶点入度减 1；

如果边的终顶点入度减至 0，则该顶点进入栈；

⑶ 如果输出顶点个数少于 AOV 网络的顶点个数，说明网络中存在有向环。

在具体的算法实现中，我们使用每个顶点的 application 成员变量指向一个 Integer 对象，它表示顶点在算法执行中当前的入度。在拓扑排序过程中对顶点成员变量 application 的操作方法见代码 7-8。

代码 7-8　拓扑排序算法中，对 v.application 的操作

```
//取或设置顶点 v 的当前入度
private int getTopInDe(Vertex v){
    return ((Integer)v.getAppObj()).intValue();
}
private void setTopInDe(Vertex v, int indegree){
    v.setAppObj(Integer.valueOf(indegree));
}
```

算法 7-6 给出了拓扑排序的具体实现。

算法 7-6 toplogicalSort

输入：AOV 网络
输出：拓扑序列
代码：

```
public Iterator toplogicalSort(){
    LinkedList topSeq = new LinkedListDLNode();  //拓扑序列
    Stack s = new StackSLinked();
    Iterator it = getVertex();
    for(it.first(); !it.isDone(); it.next()){          //初始化顶点集应用信息
        Vertex v = (Vertex)it.currentItem();
        v.setAppObj(Integer.valueOf(v.getInDeg()));
        if (v.getInDeg()==0) s.push(v);
    }
    while (!s.isEmpty()){
        Vertex v = (Vertex)s.pop();
        topSeq.insertLast(v);                          //生成拓扑序列
        Iterator adjIt = adjVertexs(v);                //对于 v 的每个邻接点入度减 1
        for(adjIt.first(); !adjIt.isDone(); adjIt.next()){
            Vertex adjV = (Vertex)adjIt.currentItem();
            int in = getTopInDe(adjV)-1;
            setTopInDe(adjV, in);
            if (in==0) s.push(adjV);                   //入度为 0 的顶点入栈
        }//for adjIt
    }//while
    if (topSeq.getSize()<getVexNum()) return null;
    else return topSeq.elements();
}
```

算法 7-6 说明：算法初始化部分需要 O(|V|)时间；在算法过程中，每个顶点最多均入栈、出栈一次，需要 O(|V|)时间；顶点入度减 1 的操作在整个算法中共执行|E|次；因此算法总的时间复杂度为 O(|V|+|E|)。

4.10.2关键路径

与 AOV 网络对应的是边表示活动的 AOE 网络。如果在有向无环的带权图中：

- 用有向边表示一个工程中的各项活动(activity)；
- 用边上的权值表示活动的持续时间(duration)；
- 用顶点表示事件(event)；

则这样的有向图叫做用边表示活动的网络，简称 **AOE (activity on edges)网络**。AOE 网络在某些工程估算方面非常有用。例如，AOE 网络可以使人们了解：(1) 完成整个工程至少需要多少时间。(2) 为缩短完成工程所需的时间，应当加快哪些活动。

例如 7.7.1 小节中图 7-21 用 AOV 网络表示的工程，可以用图 7-22 所示的 AOE 网络表示，并假设活动 a 需时 3 天、活动 b 需时 1 天、活动 c 需时 2 天、活动 d 需时 5 天、活动 e 需时 2 天。在图中我们可以看到，从工程开始到结束，总共至少需要 10 天时间，并且如果能够减少完成活动 a、d、e 所需的时间，则完成整个工程所需的时间可以减少。

由于一个工程只有一个开始点和一个完成点，所以在正常情况下，AOE 网络中只有一

个入度为 0 的顶点，也只有一个出度为 0 的顶点，它们分别称之为源点和汇点。

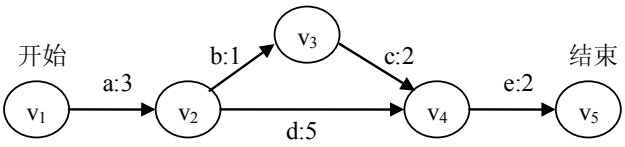


图 7-22 AOE 网

通过上面的例子，我们看到：在AOE网络中, 有些活动顺序进行，有些活动并行进行。从源点到各个顶点，以至从源点到汇点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做**关键路径（critical path）**。例如图 7-22 所示的AOE网络中的关键路径为ρ = (v_1 , v_2 , v_4 , v_5)，其长度为 10。需要注意的是关键路径可能不只一条。

要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到关键路径。例如图 7-22 所示的 AOE 网络中的关键活动是 a、d、e，找到这些关键活动就找到了关键路径。

在图G = {V , E}中，假设V = { v_0 , v_1 , … , v_{n-1}}，其中n = |V|，v_0是源点，v_{n-1}是汇点。为求关键活动，我们定义以下变量：

- 事件v_i的最早可能开始时间**Ve[i]**：是从源点v_0到顶点v_i的最长路径长度。
- 活动a_k的最早可能开始时间**e[k]**：设活动a_k在边<v_i , v_j>上，则e[k]是从源点v_0到顶点v_i的最长路径长度。因此，e[k] = Ve[i]。
- 事件v_i的最迟允许开始时间**Vl[i]**：是在保证汇点v_{n-1}在Ve[n-1]时刻完成的前提下，事件v_i允许的最迟开始时间。
- 活动a_k的最迟允许开始时间**l[k]**：设活动a_k在边<v_i , v_j>上，l[k]是在不会引起时间延误的前提下，该活动允许的最迟开始时间。l[k] = Vl[j] - dur(<i , j>)。其中，dur(<i , j>) = weight(<v_i , v_j>)是完成a_k所需的时间。
- 时间余量 **l[k] - e[k]**：表示活动a_k的最早可能开始时间和最迟允许开始时间的时间余量。l[k] = e[k]表示活动a_k是没有时间余量的关键活动。

为找出关键活动，需要求各个活动的e[k]与l[k]，以判别是否 l[k] = e[k]。为求得e[k]与l[k]，需要先求得从源点v_0到各个顶点v_i的Ve[i]和Vl[i]。

为求 Ve[i]和 Vl[i]需分两步进行：

⑴ 从 Ve[0] = 0 开始向汇点方向推进

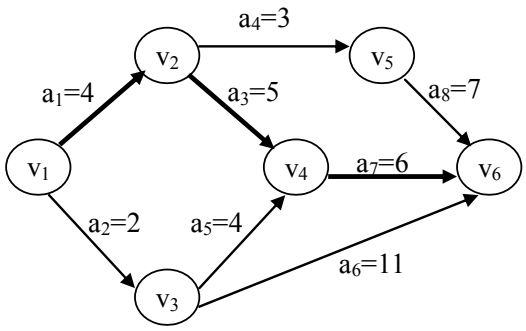
Ve[j] = Max{ Ve[i] + dur(<i , j>)| v_i是v_j的所有直接前驱顶点}

⑵ 从 Vl[n-1] = Ve[n-1]开始向源点方向推进

Vl[i] = Min{ Vl[j] - dur(<i , j>)| v_j是v_i的所有直接后续顶点}

这两个递推公式的计算必须分别在拓扑有序和逆拓扑有序的前提下进行。也就是说 Ve[j]必须在其所有直接前驱顶点的最早开始时间求得之后才能进行；Vl[i]必须在其所有直接后续顶点的最迟开始时间求得之后才能进行。因此，可以在拓扑序列的基础上求解关键活动。

在图 7-23（a）所示的AOE网络上求关键活动的过程如图 7-23（b）。在图 7-23（b）中首先求出了各个顶点的最早开始时间和最迟开始时间，然后求出各个活动的最早开始时间和最迟开始时间，最后活动时间余量为 0 的活动即为关键活动，由关键活动组成的路径是关键路径，即ρ = (v_1 , v_2 , v_4 , v_6)为关键路径。



（a）

顶点	Ve	Vl	活动	e	l	l-e
v_1	0	0	a_1	0	0	**0**
v_2	4	4	a_2	0	2	2
v_3	2	4	a_3	4	4	**0**
v_4	9	9	a_4	4	5	1
v_5	7	8	a_5	2	5	3
v_6	15	15	a_6	2	4	2
			a_7	9	9	**0**
			a_8	7	8	1

（b）

图 7-23 关键活动

下面我们给出求关键路径的算法：

⑴ 对图中的顶点进行拓扑排序，求出拓扑序列与逆拓扑序列；若拓扑序列中顶点数少于|V|，说明图中有环，返回；

⑵ Ve[0] = 0，在拓扑序列上求各顶点最早开始时间；

⑶ Vl[n-1] = Ve[n-1]，在逆拓扑序列上求各顶点最迟开始时间；

⑷ 遍历图中所有边<u , v>∈E，判断其是否为关键活动。

在算法的具体实现中采用如下策略：首先，各个顶点 v 的 Ve[v]、Vl[v]等信息使用顶点的 application 成员变量存储，此时 application 指向 Vtime 类的对象，在算法中对 application 的操作见代码 7-9。其次，判断边<u , v>是否为关键活动时，不用求出 l[k]与 e[k]，只用判断 Ve(u)与 Vl(v) - weight(<u , v>)是否相等即可。最后，AOE 网络关键路径可能不只一条，因此如果某边是关键活动，将该边标记为 Edge. CRITICAL 即可。

代码 7-9　求关键路径算法中，对 v.application 的操作

```
//取顶点 v 的最早开始时间与最迟开始时间
private int getVE(Vertex v){
    return ((Vtime)v.getAppObj()).getVE();
}
private int getVL(Vertex v){
    return ((Vtime)v.getAppObj()).getVL();
}
//设置顶点 v 的最早开始时间与最迟开始时间
private void setVE(Vertex v, int ve){
    ((Vtime)v.getAppObj()).setVE(ve);
}
private void setVL(Vertex v, int vl){
```

```
            ((Vtime)v.getAppObj()).setVL(vl);
        }
```

其中 Vtime 类的定义见代码 7-10

代码 7-10　Vtime 类定义

```
public class Vtime {
    private int ve; //最早发生时间
    private int vl;  //最迟发生时间
    //构造方法
    public Vtime() {     this(0,Integer.MAX_VALUE); }
    public Vtime(int ve, int vl){
        this.ve = ve;
        this.vl = vl;
    }
    //get&set method
    public int getVE(){ return ve;}
    public int getVL(){ return vl;}
    public void setVE(int t){ ve = t;}
    public void setVL(int t){ vl = t;}
}
```

算法 7-7 给出了求关键路径的具体实现。

算法 7-7 toplogicalSort

输入：AOE 网络

输出：标记关键路径

代码：

```
    public void criticalPath(){
        Iterator it = toplogicalSort();
        resetEdgeType();                          //重置图中各边的类型信息
        if (it==null) return;
        LinkedList reTopSeq = new LinkedListDLNode();          //逆拓扑序列
        for(it.first(); !it.isDone(); it.next()){     //初始化各点 ve 与 vl，并生成逆拓扑序列
            Vertex v = (Vertex)it.currentItem();
            Vtime time = new Vtime(0,Integer.MAX_VALUE);    //ve=0,vl=∞
            v.setAppObj(time);
            reTopSeq.insertFirst(v);
        }
        for(it.first(); !it.isDone(); it.next()){     //正向拓扑序列求各点 ve
            Vertex v = (Vertex)it.currentItem();
            Iterator adjIt = adjVertexs(v);
            for(adjIt.first(); !adjIt.isDone(); adjIt.next()){
                Vertex adjV = (Vertex)adjIt.currentItem();
                Edge e = edgeFromTo(v,adjV);
                if (getVE(v)+e.getWeight()>getVE(adjV))          //更新最早开始时间
                    setVE(adjV, getVE(v)+e.getWeight());
            }
```

```
        }
        Vertex dest = (Vertex)reTopSeq.first().getData();
        setVL(dest, getVE(dest));                    //设置汇点 vl=ve
        Iterator reIt = reTopSeq.elements();
        for(reIt.first(); !reIt.isDone(); reIt.next()){    //逆向拓扑序列求各点 vl
            Vertex v = (Vertex)reIt.currentItem();
            Iterator adjIt = adjVertexs(v);
            for(adjIt.first(); !adjIt.isDone(); adjIt.next()){
                Vertex adjV = (Vertex)adjIt.currentItem();
                Edge e = edgeFromTo(v,adjV);
                if (getVL(v)>getVL(adjV)-e.getWeight())    //更新最迟开始时间
                    setVL(v, getVL(adjV)-e.getWeight());
            }
        }
        Iterator edIt = edges.elements();
        for(edIt.first(); !edIt.isDone(); edIt.next()){              //求关键活动
            Edge e = (Edge)edIt.currentItem();
            Vertex u = e.getFirstVex();
            Vertex v = e.getSecondVex();
            if (getVE(u)==getVL(v)-e.getWeight()) e.setToCritical();
        }
    }
```

算法 7-7 说明：算法首先调用拓扑排序算法得到拓扑序列，需要时间O(|V|+|E|)；然后初始化各顶点 Ve 和 Vl 并生成逆拓扑序列，需要时间O(|V|)；在正向拓扑序列求各顶点 ve 过程中对所有顶点和边访问一次，需时O(|V|+|E|)，逆向拓扑序列求各顶点 vl 需要同样的时间O(|V|+|E|)；最后遍历所有的边判断其是否为关键路径，需时O(|E|)；因此，算法总的时间复杂度为O(|V|+|E|)。

第八章　查找

在非数值运算问题中，数据存储量一般很大，为了在大量信息中找到某些值，需要用到查找技术，为了提高查找效率，需要对一些数据进行排序。查找和排序的数据处理量占有非常大的比重，故查找和排序的有效性直接影响到算法的性能，因而查找和排序是重要的处理技术。从本章开始，我们将介绍查找和排序。

8.1 查找的定义

8.1.1 基本概念

首先介绍几个与查找有关的基本概念。

查找表（search table）是由同一类型的数据元素（或**记录**）构成的集合。由于数据元素之间存在着完全松散的关系，因此查找表是一种非常灵活的结构，可以利用任意数据结构实现。

关键字（key）是数据元素的某个数据项的值，用它可以标识查找表中一个或一组数据元素。如果一个关键字可以唯一标识查找表中的一个数据元素，则称其为**主关键字**，否则为**次关键字**。当数据元素仅有一个数据项时，其关键字即为该数据元素的值。

查找（search）根据给定的关键字值，在查找表中确定一个关键字与给定值相同的数据元素，并返回该数据元素在查找表中的位置。若找到相应数据元素，则称查找成功，否则称查找失败，此时返回空地址。

对于查找表的查找，一般有两种情况：一种是**静态查找**，指在查找过程中只是对数据元素进行查找；另一种是**动态查找**，指在实施查找过程中，插入查找失败的元素，或从查找表中删除已查到的某个元素，即允许表中元素发生变化。

平均查找长度（average search length , ASL）是为确定数据元素在查找表中的位置，需要和给定的值进行比较的关键字个数的期望值，称为查找算法在查找成功时的平均查找长度。

对于长度为 n 的查找表，查找成功时的平均查找长度为

$$ASL = P_1C_1 + P_2C_2 + \ldots + P_nC_n = \sum_{i=1}^{n} P_iC_i$$

其中：P_i为查找查找表中第i个数据元素的概率，C_i为找到表中第i个数据元素时，已经进行的关键字的比较次数。

需要注意，这里讨论的平均查找长度是在查找成功的情况下进行的讨论，换句话说，我们认为每次查找都是成功的。前面提到查找可能成功也可能失败，但是在实际应用的大多数情况下，查找成功的可能性要比不成功的可能性大得多，特别是查找表中数据元素个数 n 较大时，查找不成功的概率可以忽略不计。由于查找算法的基本运算是关键字之间的比较操作，所以平均查找长度可以用来衡量查找算法的性能。

在一个结构中查找某个数据元素的过程依赖于这个数据元素在结构中所处的位置。因此，对表进行查找的方法取决于表中数据元素以何种关系组织在一起，该关系是为进行查找

人为加在数据元素上的。为此查找有基于线性结构的查找还有基于树结构的查找，而这些查找都是基于关键字的比较进行的，都属于比较式的查找；另一类查找法是计算式查找法，也称为 HASH 查找法。

8.1.2 查找表接口定义

从上述定义中我们看到，在查找表中除了可以完成查找操作，还可以动态的改变查找表中的数据元素，即可以进行插入和删除的操作。为此，下面给出查找表的接口定义。

代码 8-1 查找表接口定义

```
public interface SearchTable {
    //查询查找表当前的规模
    public int getSize();
    //判断查找表是否为空
    public boolean isEmpty();
    //返回查找表中与元素 ele 关键字相同的元素位置；否则，返回 null
    public Node search(Object ele);
    //返回所有关键字与元素 ele 相同的元素位置
    public Iterator searchAll(Object ele);
    //按关键字插入元素 ele
    public void insert(Object ele);
    //若查找表中存在与元素 ele 关键字相同元素，则删除一个并返回；否则，返回 null
    public Object remove(Object ele);
}
```

在这里需要注意的是，在查找表接口定义并没有使用各个数据元素的关键字作为查找或删除的输入，而是以数据元素本身作为输入参数，这是考虑到关键字本身是数据元素的一部分，使用第三章中代码 3-3 定义的 Strategy 接口即可完成对两个数据元素的按关键字的判等和比较。因此，在查找表的具体实现中引入具体实现了 Strategy 接口的类的对象就可以完成按照元素关键字进行的比较等操作。

8.2 顺序查找与折半查找

基于线性结构的查找主要介绍两种最常见的查找方法：顺序查找和折半查找。

- **顺序查找**

顺序查找的特点是，用所给的关键字与线性表中各元素的关键字逐个进行比较，直到成功或失败。

算法的基本思想是：在查找表的一端设置一个称为“监视哨”的附加单元，存放要查找的数据元素关键字。然后从表的另一端开始查找，如果在“监视哨”位置找到给定关键字，则失败，否则成功返回相应元素的位置。

该算法思想与第三章中线性表的 indexOf 方法一致。只是在这里进行查找之前在表的一端设置了一个监视哨，其目的在于免去查找过程中每一步都检测整个表是否查找完毕。使用这一程序设计技巧，可以使得顺序查找在一个成功的检索中，比较次数大于等于 6 时就是一个改进。实践证明，在查找表的规模 $n \geq 1000$ 时，进行一次查找所需的平均时间几乎减少一

半。该算法较为简单，请读者自行实现。

【查找分析】这里，我们用平均查找长度（ASL）分析顺序查找算法的性能。假设查找表长度为n，查找每个数据元素的概率相等，均为 1/n。并且将监视哨设置在高端，那么查找第i个数据元素时需要进行i次比较，即$C_i = i$，则平均查找长度为

$$ASL = \sum_{i=1}^{n} P_i C_i = (1 + 2 + \ldots + n)/n = (n+1)/2$$

顺序查找和我们后面将要介绍的其他查找算法相比，其缺点是平均查找长度较大，特别是当n较大时，查找效率较低。然而，它的优点是：算法简单且适应面广。它对查找表的结构没有要求，无论数据元素是否按关键字有序[①]均可适用。

■ 折半查找

折半查找又称为二分查找，这种查找方法需要待查的查找表满足两个条件：首先，查找表必须使用顺序的存储结构；其次，查找表必须按关键字大小有序排列。

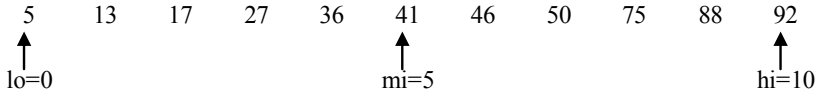
算法的基本思想是：首先，将查找表中间位置数据元素的关键字与给定关键字比较，如果相等则查找成功；否则利用中间元素将表一分为二，如果中间元素关键字大于给定关键字，则在前一子表中进行折半查找，否则在后一子表中进行折半查找。重复以上过程，直到找到满足条件的元素，则查找成功；或直到子表为空为止，此时查找不成功。

例 8-1 已知如下 11 个数据元素的有序表，在有序表中查找关键字 27 和 78。

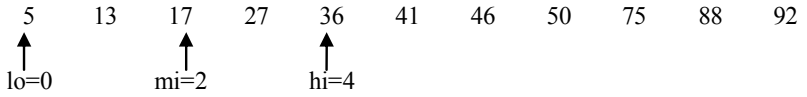
(5 , 13 , 17 , 27 , 36 , 41 , 46 , 50 , 75 , 88 , 92)

假设指针 lo 和 hi 分别指示待查元素所在范围的下界和上界，指针 mi 指示查找区间的中间位置，$mi = \lfloor (lo+hi)/2 \rfloor$。在这里，lo 和 hi 的初始值分别为 0 和 10，即查找范围为[0 , 10]。

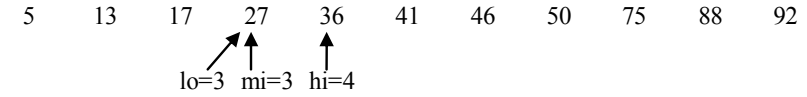
下面先说明给定关键字 27 的查找过程：



首先，将给定关键字 27 与 mi 所指元素关键字 41 进行比较，结果是中间元素的关键字大，因此下一步在[lo , mi-1]的范围内继续查找。因此，令指针 hi = mi – 1 = 4，并重新求得 $mi = \lfloor (0+4)/2 \rfloor = 2$。



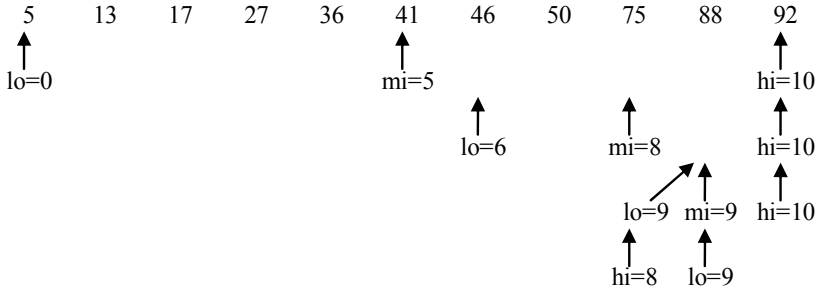
然后，仍然用给定关键字 27 与 mi 所指元素关键字 17 进行比较，结果是 mi 所指元素的关键字小，因此下一步在[mi+1 , hi]范围内继续查找。因此，令指针 lo= mi + 1 = 3，并重新求得 $mi = \lfloor (3+4)/2 \rfloor = 3$。



此时，再将给定关键字 27 与 mi 所指元素关键字 27 进行比较，结果相等，则查找成功，返回 mi 的值即为需要查找元素在表中的位置。

再看 key = 78 的查找过程：

[①] 如果表中所有元素的关键字按递增顺序排列，则称表中元素按关键字有序；反之称为表中元素按关键字逆向有序。



此时，因为下界 lo 大于上界 hi，则说明表中没有关键字为 78 的元素，查找失败。

下面我们给出一个在有序整型数组上的折半查找算法的实现。

算法 8-1 binSearch

输入：整型数组 s，查找范围 low、high，待查关键字 key

输出：查找结果在 s 中的位置

代码：

```
public int binSearch(int[] s, int low, int high, int key){
    while(low<=high){
        int mid = (low + high)/2;
        if (s[mid]==key) return mid;
        else if (s[mid]>key) high = mid - 1;
        else low = mid + 1;
    }
    return -1;
}
```

以上是折半查找算法的非递归形式，折半查找算法也可以写成递归的形式，并且可以将这一算法扩展到 Object 类型的数据元素。相关程序请读者自行设计实现。

【查找分析】从折半查找过程看，以表的中间元素为比较对象，并以中间元素将表分割为两个子表，对定位到的子表继续这种操作。所以，对表中每个数据元素的查找过程，可用二叉树来描述，称这个描述查找过程的二叉树为判定树。

例如，例 8-1 中查找表所对应的判定树如图 8-1 所示。树中每个结点表示查找表中一个数据元素在表中的位置，结点旁边的数字是每个元素的值。从判定树上可见，查找关键字 27 的过程正好是走了一条从根结点到与 27 对应的结点的路径，和给定关键字比较的次数为该路径上的结点数。类似的，找到有序表中任意元素的过程都是走了一条从根到与该元素对应结点的路径，和给定关键字比较的次数就是该结点的层次数加 1。

由于判定树的叶子结点所在层次相差不超过 1，故虽然判定树不是完全二叉树，但具有 n 个结点的判定树的深度与 n 个结点的完全二叉树的深度相同，均为$\lfloor \log n \rfloor$。折半查找在查找成功时进行比较的关键字个数不超过树的深度加 1，即$\lfloor \log n \rfloor+1$。在查找失败时，折半查找走了一条从根结点到空结点的路径，比较的关键字个数是该路径上非空结点的个数，因此，折半查找在查找失败时进行比较的关键字个数最多也不超过树的深度加 1，也为$\lfloor \log n \rfloor+1$。

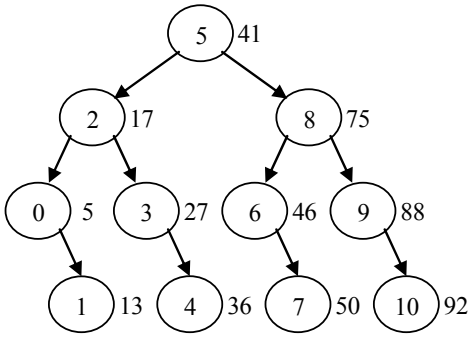


图 8-1 例 8-1 的判定树

下面来分析折半查找的平均查找长度。为简化讨论，假设表的长度$n = 2^{h+1}-1$，则相应

判定树必为深度为h的满二叉树。又假设每个数据元素的查找概率相等，均为 1/n，则折半查找的平均查找长度为：

$$ASL=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{j=0}^{h}(j+1)\times 2^j$$

令 $$s=\sum_{j=0}^{h}(j+1)\times 2^j=1\cdot 2^0+2\cdot 2^1+3\cdot 2^2+...+(h+1)\cdot 2^h$$

则 $$2s=\sum_{j=0}^{h}(j+1)\times 2^{j+1}=1\cdot 2^1+2\cdot 2^2+3\cdot 2^3+...+(h+1)\cdot 2^{h+1}$$

故 $$2s-s=(h+1)\cdot 2^{h+1}-\sum_{j=0}^{h}2^j=(h+1)\cdot 2^{h+1}-(2^{h+1}-1)=(n+1)\log(n+1)\quad -n$$

将 s 带入 ASL，得 $ASL=\frac{n+1}{n}\times\log(n+1)\ -1\approx\log(n+1)\ -1$

可见，折半查找的效率比顺序查找高，但折半查找只适合于有序表，且限于顺序存储结构，插入删除困难。因此折半查找适用于不经常变动而查找频繁的有序表。

8.3 查找树

基于树的查找方法是将待查表组织成特定的树结构，并在树结构的基础上实现查找的方法。下面我们主要介绍二叉查找树、平衡二叉树和 B-树。

8.3.1 二叉查找树

所谓**二叉查找树（binary search tree , BST）**或者是一棵空树；或者是具有以下性质的二叉树：

⑴ 若它的左子树不空，则其左子树中所有结点的值不大于根结点的值；

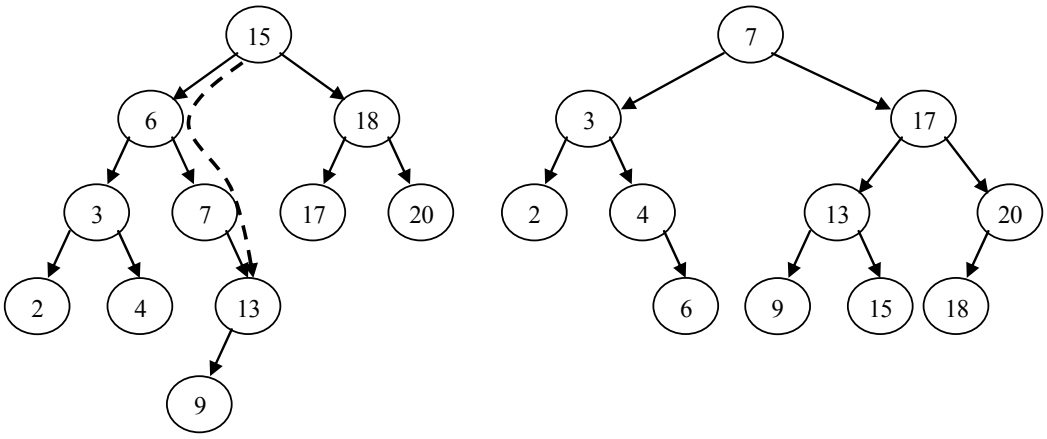
⑵ 若它的右子树不空，则其右子树中所有结点的值不小于根结点的值；

⑶ 它的左、右子树都是二叉查找树。

如图 8-2（a）所示就是一棵二叉查找树。需要注意的是，在这里并不要求所有结点元素的关键字必须互异。

分析二叉查找树的结构和二叉树的中序遍历方法，我们有以下结论：中序遍历一棵二叉查找树可以得到一个按关键字递增的有序序列。例如对图 8-2（a）中的二叉查找树进行中序遍历，得到的遍历序列为：2 , 3 , 4 , 6 , 7 , 9 , 13 , 15 , 17 , 18 , 20。

下面分析在二叉查找树中进行查找以及动态修改查找表的算法。由于二叉查找树是二叉树，所以实现二叉查找树的类 BSTree 可以通过继承第六章中实现的二叉树来完成。在以下算法的实现中涉及了继承自 BinaryTreeLinked 类的两个成员变量：一个是二叉树结点类型的 root 变量，它指向二叉查找树的根结点；另一个是第三章中定义的 Strategy 接口类型变量 strategy，用于完成数据元素之间的比较操作；同时在二叉查找树中定义了一个新的二叉树结点类型的成员变量 startBN，它的作用是用于平衡二叉树中实现平衡化操作而设，它的作用和含义详见 8.3.2 小节。



（a）　　（b）

图 8-2 二叉查找树

■ 查找算法

二叉查找树的基本查找方法是从根结点开始，递归的缩小查找范围，直到发现目标元素为止（查找成功）或查找范围缩小为空树（查找失败）。

查找算法的基本思想是：若查找树不为空，将待查关键字与根结点元素关键字比较，若相等则返回根结点；否则判断待查关键字与根结点关键字的大小，如果待查关键字小，则递归的在查找树的左子树中查找，否则递归的在查找树的右子树中查找。

算法 8-2 实现了在二叉查找树中查找数据元素的算法。

算法 8-2 search

输入：待查元素 ele

输出：对应元素在二叉查找树中的结点位置

代码：

```
public Node search(Object ele){
    return binTSearchRe (root, ele);
}
private Node binTSearchRe(BinTreeNode rt, Object ele){
    if (rt==null) return null;
    switch(strategy.compare(ele,rt.getData())){
        case 0:    return rt;                                    //等于
        case -1: return binTSearchRe(rt.getLChild(),ele); //小于
        default: return binTSearchRe(rt.getRChild(),ele); //大于
    }
}
```

算法 8-2 说明：递归算法 binTSearchRe 从根结点 root 开始，算法在递归的执行过程中只会沿着 case 语句的一条分支进行；查找成功时，实际上就是走了一条从根结点到某个结点的路径，路径上结点的个数为算法执行中进行关键字比较的次数；查找失败时，走了一条从根到某个空结点的路径，算法中进行关键字的比较次数依然是路径上结点个数；因此算法的时间复杂度为 O(h)，h 为二叉查找树的高度。

由于在二叉查找树中查找数据元素，实际上是判断当前结点，若不等则向左或右子树不断深入的过程。因此也可以使用算法 8-3 所示的非递归形式实现。例如在图 8-2（a）所示的二叉查找树中查找元素 13，就从根结点 15 开始向左、右、右走了一条到达 13 的路径，在图中用虚线表示。

算法 8-3 binTSearch

输入：根结点 rt，待查元素 ele

输出：对应元素在 rt 为根的二叉查找树中的结点位置

代码：

```
private Node binTSearch(BinTreeNode rt, Object ele){
    while(rt!=null){
        switch(strategy.compare(ele,rt.getData())){
            case 0:    return rt;                    //等于
            case -1: rt = rt.getLChild(); break; //小于
            default: rt = rt.getRChild();          //大于
        }
    }
    return null;
}
```

算法 8-3 在查找的过程中也是走了从根到某个结点的一条路径，因此算法的时间复杂度与算法 8-2 一样，也是 O(h)。

【查找分析】通过图 8-2（a）以及上面的查找算法分析，我们知道二叉查找树的关键字比较次数不超过树的高度。然而，和折半查找不同的是，对长度为 n 的查找表进行折半查找，其的判定树是唯一的；而含有 n 个结点的二叉查找树却不唯一。图 8-2（a）与 8-2（b）的两棵二叉查找树中结点的个数和值都相同，但一个的高度为 4，一个高度为 5。假设每个元素的查找概率相等，均为 1/11，则（a）树的平均查找长度为

$$ASL_{(a)} = (1+2+2+3+3+3+3+4+4+4+5)/11 = 34/11$$

而（b）树的平均查找长度为

$$ASL_{(b)} = (1+2+2+3+3+3+3+4+4+4+4)/11 = 33/11$$

因此，含有n个结点的二叉查找树的平均查找长度和树的形态有关。假设二叉查找树中每个结点的关键字互异。在具有n个结点的二叉树中，树的最小高度为$\log n$ ，即在最好的情况下二叉查找树的平均查找长度与折半查找一样与$\log n$ 成正比。具有n个结点的二叉树可以退化为一个单链表，其深度为n-1，此时其平均查找长度为(n+1)/2，与顺序查找相同，这是最差的情况。在平均情况下，如果随机生成二叉查找树，其平均查找长度和$\log n$ 是等数量级的[①]。

■ 最大最小值

在二叉查找树中，最小元素总是能够通过根结点向左不断深入，直到到达最左的一个叶子结点找到；而最大元素总是能够通过根结点向右不断深入，直到到达最右的一个叶子结点找到。例如图 8-2 所示。下面的算法 8-4 和算法 8-5 分别返回指向给定根结点 v 的二叉查找树中最大和最小元素所在的结点。

算法 8-4 min

输入：根结点 v

输出：在 v 为根的二叉查找树中最小元素的位置

代码：

```
public Node min(BinTreeNode v){
    if (v!=null)
        while (v.hasLChild()) v = v.getLChild();
```

[①] 可以证明在平均情况下，当n≥2 时，二叉查找树的平均查找长度$ASL \le 2\left(1+\frac{1}{n}\right)\ln n$ 。

```
        return v;
    }
```

算法 8-5 max

输入：根结点 v

输出：在 v 为根的二叉查找树中最大元素的位置

代码：

```
    public Node max(BinTreeNode v){
        if (v!=null)
            while (v.hasRChild()) v = v.getRChild();
        return v;
    }
```

算法 8-4 和算法 8-5 的实现也是走了一条从根结点到叶子结点的路径，因此，算法的时间复杂度和查找算法一样也是 O(h)，h 是以 v 为根的树的高度。

■ 前驱和后续

对于二叉查找树中某个给定结点 v，在有些情况下确定该结点在中序遍历序列中的直接前驱和后续是很重要的。如果查找树中结点关键字互异，那么其中序序列是一个严格递增的序列，v 的后续是比 v 大的元素中关键字最小的，v 的前驱是比 v 小的元素中关键字最大的。

在二叉查找树中确定某个结点 v 的后续结点的算法思想如下：如果结点 v 有右子树，那么 v 的后续结点是 v 的右子树中关键字最小的；如果结点 v 右子树为空，并且 v 的后续结点存在，那么 v 的后续结点是从 v 到根的路径上第一个作为左孩子结点的父结点。例如图 8-2（a）中，结点 15 的后续是它的右子树中最小的元素 17；结点 4 没有右子树，而结点 3 是从结点 4 到根结点 15 的路径上第一个作为左孩子的结点，结点 3 的父结点 6 就是结点 4 的后续。

算法 8-6 getSuccessor

输入：根结点 v

输出：返回 v 在中序遍历序列中的后续结点

代码：

```
    private BinTreeNode getSuccessor (BinTreeNode v){
        if (v==null) return null;
        if (v.hasRChild()) return (BinTreeNode)min(v.getRChild());
        while (v.isRChild()) v = v.getParent();
        return v.getParent();
    }
```

算法 8-6 无论是在结点 v 的右子树中找最小结点，或者还是沿着 v 的父指针域沿路上向，算法的时间复杂度都是 O(h)，h 是二叉查找树的高度。

确定结点 v 的直接前驱结点的算法思想与确定前驱结点的算法正好对称，算法的实现见算法 8-7，且算法的时间复杂度同样是 O(h)。

算法 8-7 getPredecessor

输入：根结点 v

输出：返回 v 在中序遍历序列中的前驱结点

代码：

```
    private BinTreeNode getPredecessor(BinTreeNode v){
        if (v==null) return null;
        if (v.hasLChild()) return (BinTreeNode)max(v.getLChild());
```

```
        while (v.isLChild()) v = v.getParent();
        return v.getParent();
    }
```

■ 插入算法

为了在二叉查找树中插入一个结点，并且保持二叉查找树的特性，我们必须根据结点元素的关键字，确定结点插入的位置及方向，然后将新结点作为叶子结点插入。

为了判定新结点的插入位置，需要从根结点开始逐层深入，判断新结点关键字与各子树根结点关键字的大小，若新结点关键字小，则向相应根结点的左子树深入，否则向右子树深入；直到向某个结点的左子树深入而其左子树为空，或向某个结点的右子树深入而其右子树为空时，则确定了新结点的插入位置。图 8-3 图示了插入结点 14 的过程。

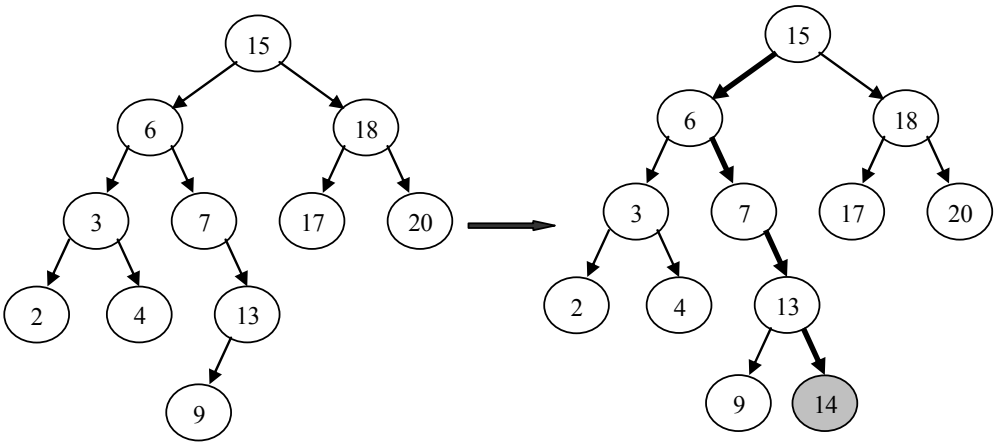


图 8-3 在二叉查找树中插入结点

算法 8-8 实现了上述操作过程。

算法 8-8 insert

输入：待插元素 ele

输出：在二叉查找树中插入 ele

代码：

```
    public void insert(Object ele){
        BinTreeNode p = null;
        BinTreeNode current = root;
        while (current!=null){                    //找到待插入位置
            p = current;
            if (strategy.compare(ele,current.getData())<0)
                current = current.getLChild();
            else
                current = current.getRChild();
        }
        startBN = p;                              //待平衡出发点 *
        if (p==null)
            root = new BinTreeNode(ele);          //树为空
        else if (strategy.compare(ele,p.getData())<0)
            p.setLChild(new BinTreeNode(ele));
        else
            p.setRChild(new BinTreeNode(ele));
```

}

算法 8-8 说明：在算法中使用了两个循环变量 current 和 p，其中 p 指向 current 的父结点。while 循环从根结点不断向下深入，当 current 为空时，该空位置为新结点的待插位置，而 p 即为新结点的父结点，判断新结点与 p 的关键字大小即可确定插入方向并完成插入操作。算法走了一条从根到某个结点 p 的一条路径，因此算法的时间复杂度为 O(h)。另外，在算法 8-8 中标记＊的一行是用来实现 AVL 树的重新平衡，在这里可以暂时不予理会。

■ 删除算法

同样在二叉查找树中删除一个特定结点 v 时，我们也要在保持二叉查找树特性的前提下进行删除。与在二叉查找树中插入结点不同的是，在二叉查找树中删除结点要复杂的多，因为在二叉查找树中插入的新结点总是作为叶子结点插入，只要找到插入位置则插入操作十分简单，而在二叉查找树中删除的结点不总是叶子结点，因此在删除一个非叶子结点时需要处理该结点的子树。

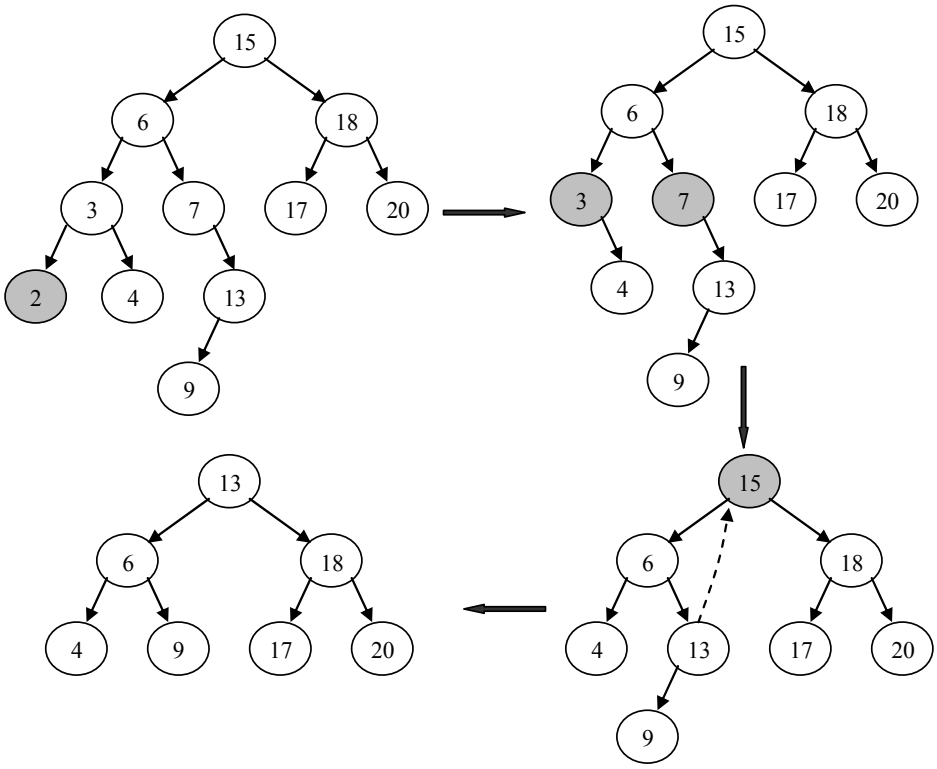


图 8-4 在二叉查找树中删除结点

下面我们分三种情况讨论结点 v 的删除：

⑴ 如果结点v为叶子结点，即其左右子树P_l和P_r均为空，此时可以直接删除该叶子结点v，而不会破坏二叉查找树的特性，因此直接从树中摘除v即可。例如在图 8-4 所示的二叉查找树中删除结点 2，由于结点 2 是叶子结点，则直接摘除即可。

⑵ 如果结点v只有左子树P_l或只有右子树P_r，此时，当结点v是左孩子时，只要令P_l或P_r为其双亲结点的左子树即可；当结点v是右孩子时，只要令P_l或P_r为其双亲结点的右子树即可。例如在图 8-4 所示的二叉查找树中删除结点 3 和 7，由于 3 是**左孩子**，因此在删除结点 3 之后，将其右子树设为其父结点 6 的**左子树**即可；同样，因为 7 是**右孩子**，因此在删除结点 7 之后，将其右子树设为其父结点 6 的**右子树**即可。

⑶ 如果结点v既有左子树P_l又有右子树P_r，此时，不可能进行如上简单处理。为了在删除结点v之后，仍然保持二叉查找树的特性，我们必须保证删除v之后，树的中序序列必须仍然有序。为此，我们可以先用中序序列中结点v的前驱或后序替换v，然后删除其前驱或后序

结点即可，此时v的前驱或后序结点必然是没有右孩子或没有左孩子的结点，其删除操作可以使用前面规定的方法完成。例如在图 8-4 所示的树中删除结点 15，由于结点 15 既有左子树又有右子树，则此时，可以先找到其前驱结点 13，并用 13 替换 15，然后删除结点 13 即可。

算法 8-9 实现了在二叉查找树中删除结点的操作。

算法 8-9 remove

输入：待删除元素 ele

输出：在二叉查找树中删除 ele

代码：

```
public Object remove(Object ele){
    BinTreeNode v = (BinTreeNode)binTSearch(root,ele);
    if (v==null) return null;              //查找失败
    BinTreeNode del = null;                //待删结点
    BinTreeNode subT = null;               //待删结点的子树
    if (!v.hasLChild()||!v.hasRChild()) //确定待删结点
        del = v;
    else{
        del = getPredecessor(v);
        Object old = v.getData();
        v.setData(del.getData());
        del.setData(old);
    }
    startBN = del.getParent();                  //待平衡出发点 *
    //此时待删结点只有左子树或右子树
    if (del.hasLChild())
        subT = del.getLChild();
    else
        subT = del.getRChild();
    if (del==root) {                            //若待删结点为根
        if (subT!=null) subT.sever();
        root = subT;
    } else
    if (subT!=null){
        //del 为非叶子结点
        if (del.isLChild()) del.getParent().setLChild(subT);
        else del.getParent().setRChild(subT);
    }
    else//del 为叶子结点
        del.sever();
    return del.getData();
}
```

算法 8-9 说明：在算法 8-9 中首先查找待删结点，如果找到则判断结点 v 的特性，如果 v 既有左子树又有右子树，则调用 getPredecessor 确定其前驱结点 pre，交换 v 与 pre 的数据，此时 pre 为待删结点 del，否则找到的结点 v 即为待删结点 del。此时，del 必定只有左子树

或只有右子树，然后根据方法⑴、⑵删除 del 即可。算法执行查找的时间为 O(h)，如果待删结点 v 不为空，需时 O(h)确定 v 的前驱，其他操作常数时间即可完成，因此算法的时间复杂度为 O(h)。另外，在算法 8-9 中标记＊的一行是用来实现 AVL 树的重新平衡，在这里可以暂时不予理会。

8.3.2 AVL 树

在 8.3.1 小节中介绍的二叉查找树中，我们看到在二叉查找树中进行查找，结点的插入和删除等操作的时间复杂度都是 O(h)，其中 h 为查找树的高度。可见，二叉查找树高度直接影响到操作实现的性能，而在某些特殊的情况下二叉查找树会退化为一个单链表，如插入的结点序列本身就有序的情况下，此时各操作的效率会下降到 O(n)，其中 n 为树的规模。因此，在结点规模固定的前提下，二叉查找树的高度越低越好，从树的形态来看，也就是使树尽可能平衡。当二叉查找树的高度为 O(log n)时，此时各算法的时间复杂度均为 O(h)= O(log n)；另一方面由于具有 n 个结点的树的高度为 Ω(log n)，如此当树的高度为 O(log n)时各操作实现的效率达到最佳。

为说明二叉树的平衡性，我们定义二叉树中结点平衡因子的概念。在二叉树中，任何一个结点 v 的**平衡因子**都定义为其左、右子树的高度差。注意，空树的高度定义为-1。

在二叉查找树 T 中，若所有结点的平衡因子的绝对值均不超过 1，则称 T 为一棵 **AVL 树**。例如，图 8-2（a）所示的二叉查找树就不是 AVL 树，因为结点 7 和 15 的平衡因子分别为-2 和+2；而 8-2（b）所示的二叉查找树是 AVL 树，其中每个结点的平衡因子的绝对值均不操作过 1。

由于 AVL 上任何结点的左右子树的高度之差都不超过 1，则可以证明高度为 h 的 AVL 树至少包含 Fib(h+3)-1（其中 Fib(i)为 Fibonacci 数列的第 i 项，$i \geq 0$）个结点，如此有 n 个结点的 AVL 树的高度和 log n 是同数量级的。可见，在 AVL 树中查找、插入、删除等操作的效率就渐进复杂度而言可以达到最佳。

AVL 树也是一棵二叉查找树，因此与一般二叉查找树一样，AVL 树也是动态的，也应该支持插入、删除等操作。然而原本一棵 AVL 树在经过插入或删除之类的操作后，某些结点的高度可能会发生变化，以至于不再满足 AVL 树的条件。在这种情况下，我们需要重新调整树的结构使之重新平衡。

■ 旋转操作

在讨论调整树结构使之重新平衡的内容之前，先介绍调整树结构而不改变二叉查找树特性的手段，即旋转操作。

1. 右旋（顺时针方向旋转）

如图 8-5 所示。假设图 8-5（a）中结点 v 是结点 p 的左孩子，X 和 Y 分别是 v 的左、右子树，Z 为 p 的右子树。所谓围绕结点 p 的右旋操作，就是重新调整这些结点位置，将 p 作为 v 的右孩子，将 X 作为 v 的左子树，将 Y 和 Z 分别作为 p 的左、右子树。围绕 p 右旋的结果如图 8-5（b）所示。

2. 左旋（逆时针方向旋转）

对称的，如图 8-6 所示。假设图 8-6（a）中结点 v 是结点 p 的右孩子，Y 和 Z 分别是 v 的左、右子树，X 为 p 的左子树。所谓围绕结点 p 的左旋操作，就是将 p 作为 v 的左孩子，将 Z 作为 v 的右子树，将 X 和 Y 分别作为 p 的左、右子树。围绕 p 左旋的结果如图 8-6（b）所示。

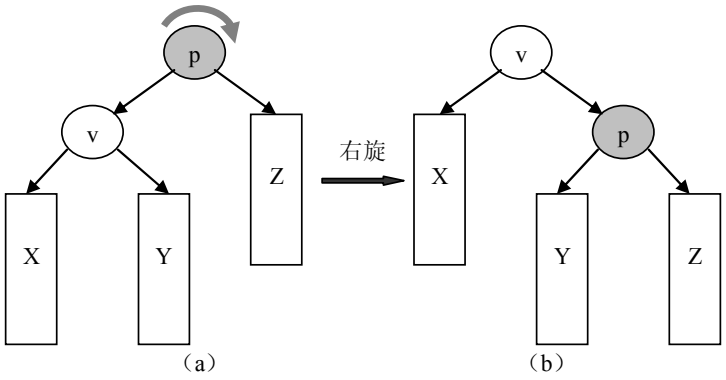


图 8-5 顺时针旋转

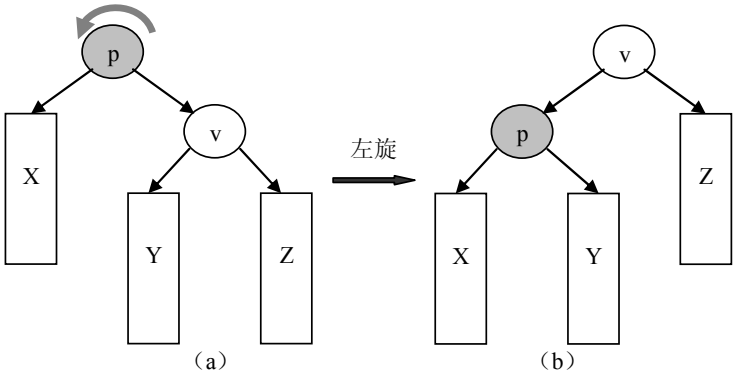


图 8-6 逆时针旋转

在以上旋转过程中，每种旋转都只涉及常数次基本操作，因此左旋和右旋均可以在 O(1) 时间内完成。而且，无论是右旋还是左旋，都不改变二叉树的中序遍历结果。例如图 8-5（a）所示二叉树的中序遍历序列为 X , v , Y , p , Z；右旋之后得到的图 8-5（b）所示二叉树的中序遍历序列仍然为 X , v , Y , p , Z。可见，如果对二叉查找树中的结点进行旋转操作不会破坏二叉查找树的特性。因此，当 AVL 树失去平衡后，我们可以采用旋转的方法，在不破坏二叉查找树特性的基础上，重新使树得到平衡。

■ 失去平衡后的重新平衡

下面首先以插入操作为例，说明 AVL 树在失去平衡后，使之重新平衡的方法。

例如，我们按照普通二叉查找树的插入算法，在图 8-7（a）所示的 AVL 树中插入结点 9。于是，如图 8-7（b）所示，结点 7 和 15 将失去平衡。

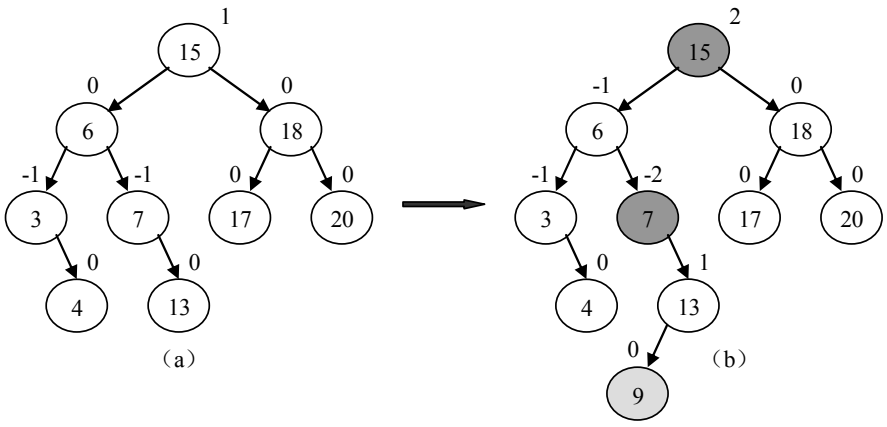


图 8-7 插入结点后 AVL 树失去平衡

一般的，若在插入新的结点 x 之后 AVL 树 T 失去平衡，则失去平衡的结点只可能是 x 的祖先，且层次数小于等于 x 的祖父的结点；也就是说失去平衡的结点是从 x 的祖父到根的路径上的结点，但这些结点并不都是失去平衡的结点，有些结点仍然可能是平衡的。例如，图 8-7（b）中，失去平衡的结点 7 和 15 是插入结点 9 的祖父到根的路径上的结点，而该路径上的另一个结点 13 仍然平衡。

为了修正失衡的现象，可以从结点 x 出发逆行向上，依次检查 x 祖先的平衡因子，找到第一个失衡的祖先结点 g。在从 x 到 g 的路径上，假设 p 为 g 的孩子结点，v 为 p 的孩子结点。根据前面的讨论，有结点 p、v 必不为空，p 至少是 x 的父亲，v 至少是 x 本身。

结点 g、p、v 之间的父子关系共有 4 种不同的组合，以下根据这 4 种不同的组合，通过对结点 g、p 的旋转，使这一局部恢复平衡。

1. p 是 g 的左孩子，且 v 是 p 的左孩子；

在这种情况下，必定是由于在 v 的后代中插入了新结点 x 而使得 g 不再平衡。针对这种情况，可以简单的通过结点 g 的**单向右旋**，即可使得以 g 为根的子树得到平衡。这一过程如图 8-8 所示。

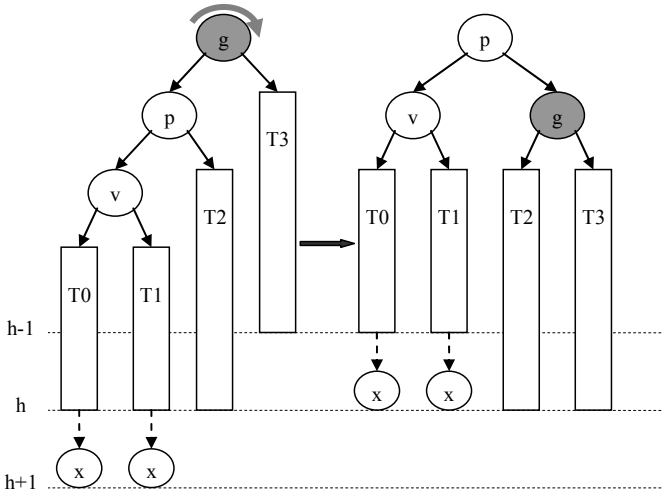


图 8-8 单向右旋

通过图 8-8 不难看出，当这一局部经过单向右旋调整后，不但失衡的结点 g 重新平衡，而且经过这一调整后，这一局部子树的高度也恢复到插入 x 以前的高度。因此 x 所有其他祖先的平衡因子也都会得到恢复，也就是说，在这种情况下，只需要一次旋转操作，即可使整棵 AVL 树恢复平衡。例如，图 8-8 中插入结点 x 之前这一局部子树的高度为 h，经过旋转操作之后，得到平衡的子树高度仍然为 h。

在插入结点 x 之前树是平衡的，此时树的高度为 O(log n)，则从 x 出发查找 g 需要花费的时间为 O(log n)，而进行平衡所需的一次旋转操作只需 O(1)时间，因此整个平衡过程只需 O(log n)时间。

2. p 是 g 的右孩子，且 v 是 p 的右孩子；

与第一种情况对称，此时，失衡是由于在 g 的右孩子的右子树中插入结点 x 造成的。在这种情况下需要通过结点 g 的**单向左旋**来完成局部的平衡。这一过程如图 8-9 所示。

通过图 8-9 不难看出，和单向右旋一样，在旋转之后不但失衡的结点 g 重新平衡，而且子树的高度也恢复到插入 x 以前的高度，因此局部的平衡能使得整棵树得到平衡。这一操作需要的时间与单向右旋相同，也为 O(log n)。

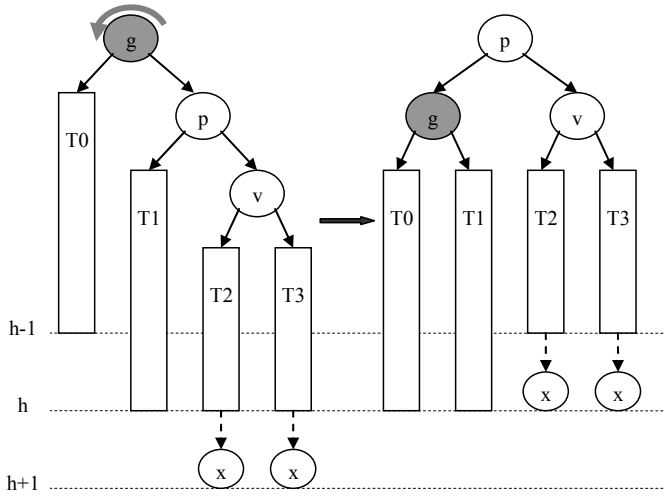


图 8-9 单向左旋

3. p 是 g 的左孩子，且 v 是 p 的右孩子；

如果是在 p 的左孩子的右子树中插入一个结点，则对应为第三种情况。在这种情况中，要使得局部的失衡重新平衡，那么需要先后进行**先左后右的双向旋转**：第一次是结点 p 的左旋，第二次是结点 g 的右旋。这一过程如图 8-10 所示。

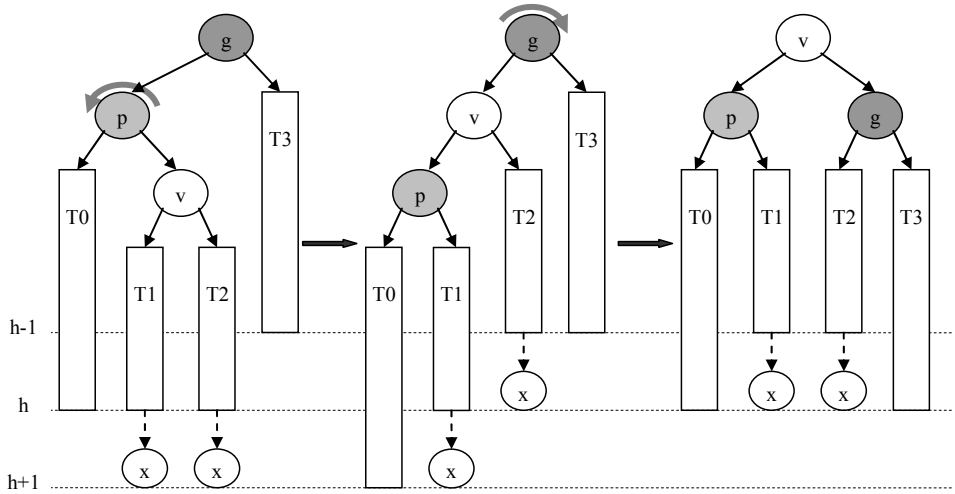


图 8-10 先左后右双向旋转

失衡的局部经过双向旋转后，失衡的结点 g 重新平衡，并且子树的高度也恢复到插入结点 x 之前的高度，结点 x 的祖先结点的平衡因子都会恢复。在这种情况下，只需要两次旋转操作就可以使得整棵 AVL 树恢复平衡。同样，为了确定 g 的位置需要 O(log n)时间，旋转操作需要 O(1)时间，因此，整个双旋操作只需 O(log n)时间。

4. p 是 g 的右孩子，且 v 是 p 的左孩子；

第四种情况与第三种情况对称，其平衡调整过程也与第三种情况对称，可以通过**先右后左的双向旋转**来完成，其中第一次是结点 p 的右旋，第二次是结点 g 的左旋。调整过程如图 8-11 所示。在旋转之后不但失衡的结点 g 重新平衡，而且子树的高度也恢复到插入 x 以前的高度，即局部的平衡能使得整棵树得到平衡。这一操作需要的时间与先左后右双向旋转相同，也为 O(log n)。

通过以上四种情况的分析，可以得到如下结论：在 AVL 树中插入一个结点后，至多只需要进行两次旋转就可以使之恢复平衡。进一步，由于在 AVL 树中按一般二叉查找树的方法插入结点需要 O(log n)时间，旋转需要 O(1)时间，则在 AVL 树中结点插入操作可以在 O(log n)时间内完成。

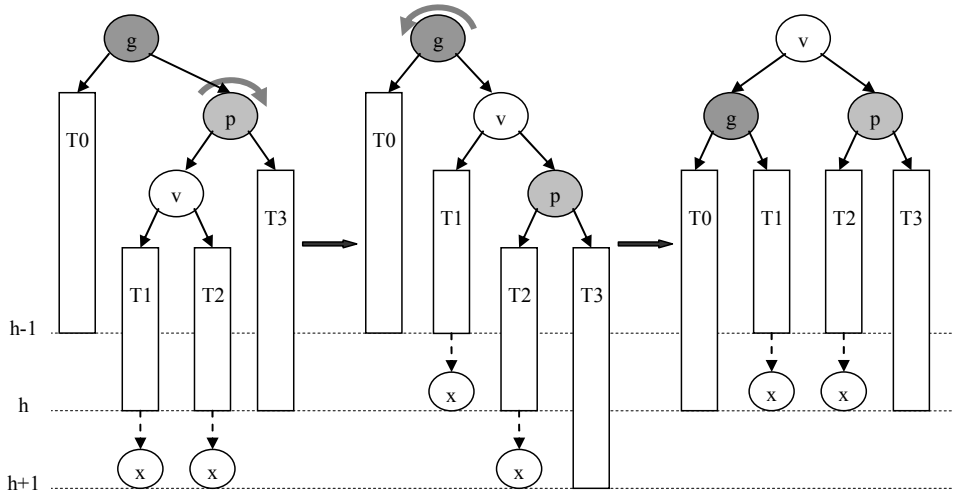


图 8-11 先右后左双向旋转

按照通常的二叉查找树删除结点的方法在AVL树中删除结点x[①]之后，如果发生失衡现象，为了重新平衡，同样从x出发逆行向上找到第一个失衡的结点g，通过旋转操作实现AVL树的重新平衡。使得结点g失衡的原因可以看成四种：

1. 失衡是由于 g 的左孩子的左子树过高造成的；
2. 失衡是由于 g 的右孩子的右子树过高造成的；
3. 失衡是由于 g 的左孩子的右子树过高造成的；
4. 失衡是由于 g 的右孩子的左子树过高造成的。

这四种情况与插入结点导致失衡的四种情况进行平衡化的旋转方法相同。但是读者需要注意两个特殊情况：其中之一是失衡可能是由于左孩子的左、右子树同时过高造成的，这种情况如图 8-12 所示。从图中可以看出对这种情况的处理可以归为情况 1 或 3 进行处理都是可以的，即图（a）可以通过简单的结点 g 的右旋完成，如图（b）所示；或通过先左后右的双向旋转完成，如图（c）所示。另一种特殊情况与此对称，同样也可以归结为情况 2 或 4 处理。

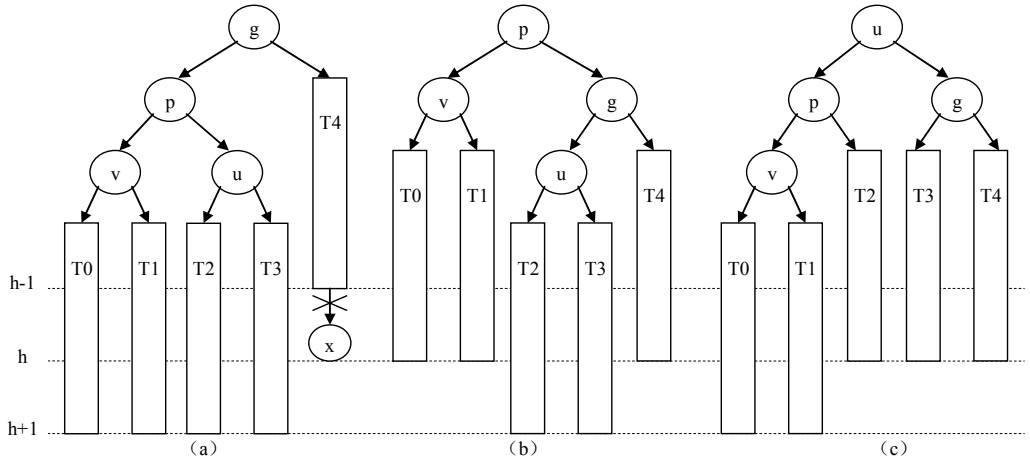


图 8-12 失衡结点左孩子的左、右子树等高的两种处理方式

删除结点后，根据失衡结点 g 的四种不同情况进行相应的旋转操作，即可使得局部恢复平衡，但是和插入结点使 AVL 树失衡不同的是，在删除结点的情况下，局部重新平衡不一定会使得 AVL 整体恢复平衡，即会造成失衡的传递。例如在图 8-12 所示的情况下，子树在删除结点之前和删除结点之后的高度都为 h+1，此时局部平衡会使 AVL 树整体平衡；然而

[①] 如果在删除x之前，曾将x与其直接前驱w交换，则以下叙述中的x实际上是w，即实际删除的结点。

在有些情况下，删除结点并调整平衡之后，局部子树的高度会降低，如图 8-13 所示。

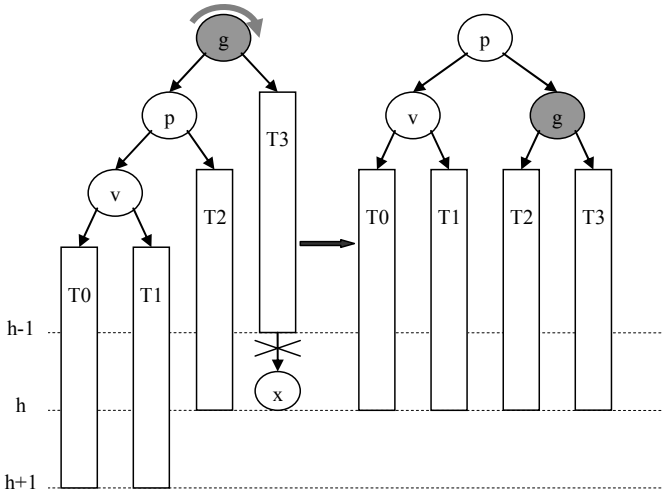


图 8-13 平衡后子树高度降低

从图 8-13 可以看到，在删除结点 x 之前，子树的高度为 h+1，而在删除结点 x 并调整平衡之后，子树的高度为 h，此时由于子树高度下降，会导致某些祖先结点将有可能因此失去平衡。这种由于对局部的重新平衡而造成更上层祖先失衡的情况，称为“失衡传播”。当发生失衡传播时，需要对到根结点路径上的失衡结点依次进行平衡操作，直到根结点为止。

由于 AVL 树的高度为 O(log n)，如此，我们有以下结论：在 AVL 树中删除一个结点后，至多需要 O(log n)次旋转即可使之平衡。进一步，由于在 AVL 树中按通常查找树的方法删除结点需要 O(log n)时间，平衡调整也需要 O(log n)时间，因此，在 AVL 树中删除结点的操作可以在 O(log n)时间内完成。

■ 旋转操作的统一实现方法[①]

前面介绍的 4 种平衡旋转操作，需要根据结点 g、p、v（p 是 g 的孩子，v 是 p 的孩子）之间的相互关系来判断应该使用的旋转方法。为此，在这里引入一种统一的算法，这种实现简洁、直观，容易实现。

在插入或删除结点 x 之后，从 x 出发逆行向上，找到第一个失衡的结点 g，然后根据 g 失衡的原因，设置 p 和 v 的指向，p 和 v 都是其父结点较高子树的根。此时，在这一局部，以 p 的兄弟为根有一棵子树，即 g 较矮的子树；以 v 的兄弟为根有一棵子树，即 p 较矮的子树；v 有两棵子树；一共有 4 棵子树。注意，在这 4 棵子树中可能有空树。显然，以失衡结点 g 为根的局部平衡是在这 3 个结点和 4 棵子树间完成的。

在确定了这 3 个结点和 4 棵子树之后，接着为它们重命名。具体的命名方法是，从左到右将 4 棵子树命名为 T0、T1、T2、T3；按照中序顺序将结点 g、p、v 分别命名为 a、b、c。四种不同旋转情况的命名见图 8-14 所示。

一旦完成重命名，无论对于那种情况，我们都能以相同的方法将这 3 个结点和 4 棵子树重新组合起来，以达到旋转平衡的目的。如图 8-14（e）所示，以结点 b 作为局部子树的根，结点 a 和 c 分别是 b 的左孩子和右孩子，结点 a 的左右子树分别是 T0、T1，结点 c 的左右子树分别是 T2、T3。

读者可将这一结果和前面的图 8-8、8-9、8-10、8-11 对照，即可看出统一算法的效果与前面的单向旋转和双向旋转完全一致。

算法 8-10 给出了旋转操作实现的统一算法。

---

① 见参考文献[]

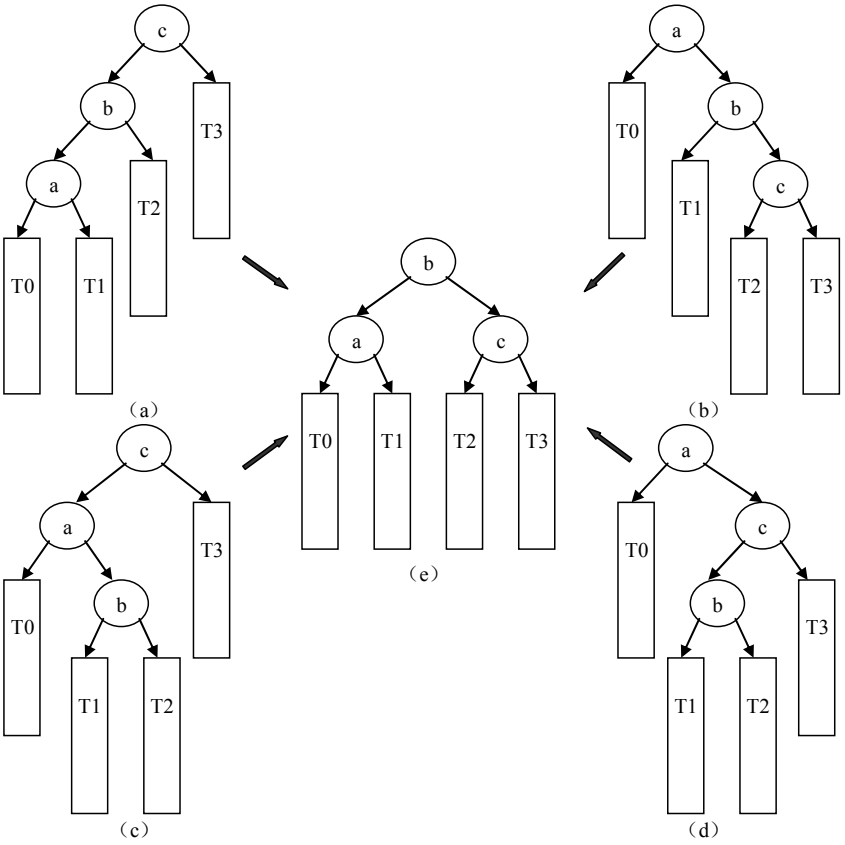


图 8-14 统一平衡方法

算法 8-10 rotate

输入：失衡的结点 z

输出：平衡后子树的根结点

代码：

```
private BinTreeNode rotate(BinTreeNode z){
    BinTreeNode  y = higherSubT(z);        //取 y 为 z 更高的孩子
    BinTreeNode  x = higherSubT(y);        //取 x 为 y 更高的孩子
    boolean isLeft = z.isLChild();         //记录：z 是否左孩子
    BinTreeNode  p = z.getParent();        //p 为 z 的父亲
    BinTreeNode  a, b, c;                  //自左向右，三个节点
    BinTreeNode  t0, t1, t2, t3;           //自左向右，四棵子树
    // 以下分四种情况重命名
    if (y.isLChild()) {                    //若 y 是左孩子，则
        c = z;      t3 = z.getRChild();
        if (x.isLChild()) {                //若 x 是左孩子(左左失衡)
            b = y;      t2 = y.getRChild();
            a = x;      t1 = x.getRChild(); t0 = x.getLChild();
        } else {                           //若 x 是右孩子(左右失衡)
            a = y;      t0 = y.getLChild();
            b = x;      t1 = x.getLChild(); t2 = x.getRChild();
        }
    } else {                               //若 y 是右孩子，则
```

```
            a = z;      t0 = z.getLChild();
            if (x.isRChild()) {                    //若 x 是右孩子(右右失衡)
                b = y;      t1 = y.getLChild();
                c = x;      t2 = x.getLChild(); t3 = x.getRChild();
            } else {                               //若 x 是左孩子(右左失衡)
                c = y;      t3 = y.getRChild();
                b = x;      t1 = x.getLChild(); t2 = x.getRChild();
            }
        }
        //摘下三个节点
        z.sever();
        y.sever();
        x.sever();
        //摘下四棵子树
        if (t0!=null) t0.sever();
        if (t1!=null) t1.sever();
        if (t2!=null) t2.sever();
        if (t3!=null) t3.sever();
        //重新链接
        a.setLChild(t0);      a.setRChild(t1);
        c.setLChild(t2);      c.setRChild(t3);
        b.setLChild(a);       b.setRChild(c);
        //子树重新接入原树
        if (p!=null)
            if (isLeft) p.setLChild(b);
            else        p.setRChild(b);
        return b;//返回新的子树根
    }
    //返回结点 v 较高的子树
    private BinTreeNode higherSubT(BinTreeNode v){
        if (v==null) return null;
        int lH = (v.hasLChild()) ? v.getLChild().getHeight():-1;
        int rH = (v.hasRChild()) ? v.getRChild().getHeight():-1;
        if (lH>rH) return v.getLChild();
        if (lH<rH) return v.getRChild();
        if (v.isLChild()) return v.getLChild();
        else return v.getRChild();
    }
```

■ AVL 树的 Java 实现

AVL 树也是一棵二叉查找树，则 AVL 树的实现可以通过继承二叉查找树来实现。AVL 树与一般二叉查找树不同的是，在插入和删除结点之后需要重新进行平衡。因此，只需要重写 insert 和 remove 方法即可。

算法 8-11 insert

输入：待插元素 ele

输出：在 AVL 树中插入 ele

代码：

```
public void insert(Object ele){
    super.insert(ele);
    root = reBalance(startBN);
}
//从 v 开始重新平衡 AVL 树
private BinTreeNode reBalance(BinTreeNode v){
    if (v==null)    return root;
    BinTreeNode c = v;
    while (v!=null) {                          //从 v 开始，向上逐一检查 z 的祖先
        if (!isBalance(v))    v = rotate(v);   //若 v 失衡，则旋转使之重新平衡
        c = v;
        v = v.getParent();                     //继续检查其父亲
    }//while
    return c;
}
//判断一个结点是否失衡
private boolean isBalance(BinTreeNode v){
    if (v==null)    return true;
    int lH = (v.hasLChild()) ? v.getLChild().getHeight():-1;
    int rH = (v.hasRChild()) ? v.getRChild().getHeight():-1;
    return (Math.abs(lH - rH)<=1);
}
```

算法 8-11 说明：首先，在 insert()方法中首先调用父类的方法完成一般二叉查找树的结点插入查找，这个过程在 AVL 树中需要 O(log n)时间；其次，通过父类中 insert 方法记录下的结点插入的位置 startBN 开始（算法 8-8 中标记＊的行），逆行向上对失衡的结点进行旋转以达到平衡的目的。

算法 8-12 remove

输入：待删元素 ele

输出：在 AVL 树中删除 ele

代码：

```
public Object remove(Object ele){
    Object obj = super.remove(ele);
    root = reBalance(startBN);
    return obj;
}
```

8.3.3 B-树

在现代计算机中通常采用分级存储系统，以最简单的二级分级存储策略为例，就是由内存储器与外存储器（磁盘）组成二级存储系统。这一策略的思想是：将最常用的数据副本存放于内存中，而大量的数据存放于外存中，借助有效的算法可以将外存的大存储量与内存高速度的优点结合起来。

一般的，在分级存储系统中，各级存储器的速度有着巨大的差异，仍然以磁盘和内存为例，前者的平均访问速度为 10ms左右，而内存储器的平均访问时间为ns级，通常在 10~100ns 左右，二者之间差异大约为 10^6。因此，为了节省一次外存储器的访问，我们宁愿多访问内存储器一百次、一千次甚至一万次。

当问题规模太大时，以至于内存储器无法容纳时，即使是前面介绍的 AVL 树，在时间上也会大打折扣。下面将要介绍的 B-树确是一种可以高效解决这个问题的一种数据结构。

所谓 m 阶 B-树或者为空树，或为满足下列特性的 m 叉树：

1. 树中每个结点至多有 m 棵子树；
2. 若根结点不是叶子结点，则至少有两棵子树；
3. 除根结点之外的所有非终端结点至少有⌈m/2⌉ 棵子树；
4. 所有的非终端结点中包含以下信息数据：（$n , A_0 , K_1 , A_1 , K_2 , \dots , K_n , A_n$）其中：$K_i$（i=1,2,…,n）为关键字，且$K_i<K_{i+1}$，$A_i$为指向子树根结点的指针（i=0,1,…,n），且指针A_{i-1}所指子树中所有结点的关键字均小于K_i （i=1,2,…,n），A_n所指子树中所有结点的关键字均大于K_n，$\lceil m/2 \rceil - 1 \le n \le m - 1$ ，n为关键码的个数。
5. 所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

例如，图 8-15 所示是一棵 4 阶 B-树。为了方便讨论，我们假设 B-树中的关键字互异。

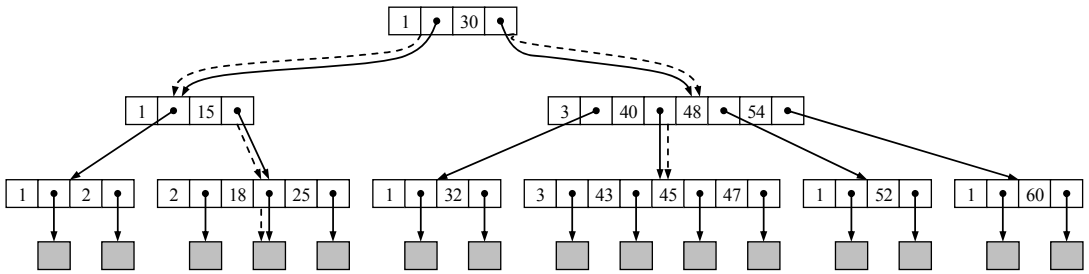


图 8-15 一棵 4 阶 B-树

■ 关键字的查找

B-树的查找类似二叉排序树的查找，所不同的是 B-树每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码，查找失败。即在 B-树上的查找过程是一个顺指针查找结点和在结点中查找关键码交叉进行的过程。例如在图 8-15 所示的 B-树中查找 23 和 47 的过程为虚线所示的路径。

正如前面所指出的，B-树的查找适合于大规模的数据。实际的做法是，将大量数据组织为一棵 B-树，并存于外存储器，B-树的根结点常驻内存。一旦需要查找，则按照上述过程，首先将根结点作为当前结点，在当前结点中顺序查找；如果当前结点不存在需要查找的关键字，则根据相应的引用，找到外存中的某一个下层结点，将其读入内存，作为新的当前结点继续查找。如此进行下去，直到相应关键字或查找失败。

由此可见，在 B-树中进行查找所需的时间，无外乎两类操作的时间消耗：一种是，在 B-树上找结点，即将外存中的结点读入内存；另一种是，在结点中找关键字。在前面曾经提到，内外存储器的平均访问时间存在巨大的差异，所以在这两部分时间中，前者必然是主要部分，后一部分时间则可以忽略。因此，B-树的查找效率取决于外存的访问次数。

【查找分析】通过以上分析，我们知道，B-树的高度是决定 B-树查找效率的首要因素 。

假设存有N个关键字的m阶B-树的高度为h，则由B-树定义：第 0 层至少有 1 个结点；第 1 层至少有 2 个结点；由于除根结点外的每个非终端结点至少有⌈m/2⌉棵子树，则第 2 层至少有 2 (⌈m/2⌉) 个结点；……；依次类推，第h–1 层至少有 2 (⌈m/2⌉) $^{h-2}$个结点，第h层至少

有 2 (⌈m/2⌉) $^{h-1}$个结点。通过观察看到叶子结点的个数恰好比关键字的个数多 1（若将所有关键字排序，那么在相邻两个关键字之间都对应一个查找失败的叶子结点，并且在第一个最小关键字之前和最后一个最大关键字之后各有一个叶子结点），而第h层的结点为叶子结点，而由此有：

$$N+1 \geq 2\,(\lceil m/2 \rceil)^{h-1}$$

则 $$k \leq \log_{\lceil m/2 \rceil}((N+1)/2)+1 = O(\log_{\lceil m/2 \rceil}(N/2))$$

例如，对于有有 1G 记录的数据，若组织成为 AVL 树，则每次查找需要 30 次外存访问，而对于 256 阶的 B-树，每次查找对应的外存访问次数约为 4~5 次。

由此，对于具有N个关键字的m阶B-树的每次查找操作，都可以在$O(\log_m N)$时间内完成。

■ 关键字的插入操作

在 B-树上插入关键字与在二叉查找树上插入结点不同，关键字的插入不是在叶子结点上进行的，而是在最底层的某个非终端结点中添加一个关键字。

为了在 m 阶 B-树中插入一个新关键字 key，首先要找到该关键字应该插入的位置，该过程实际是在 B-树中查找关键字的过程。倘若查找成功，则不再插入重复的关键字，若查找不成功，则在此查找过程中遇到的最后一个非叶子结点 p，即为关键字 key 的插入位置。然后，在结点 p 中，按照关键字有序的顺序将 key 插入。若插入后结点 p 上关键字个数不超过 m–1 个，则可直接插入到该结点上；否则，要进行调整，即结点的“分裂”。

一般情况下，结点可以如下实现“分裂”。

假设结点 p 中已有 m-1 个关键字，当插入一个关键字之后，结点中包含的信息为：

$$m\,,\,A_0\,,\,K_1\,,\,A_1\,,\,K_2\,,\,\ldots\,,\,K_m\,,\,A_m$$

此时，可以将结点 p 分裂为两个结点 u、v，其中结点 u 包含的信息为

$$\lceil m/2 \rceil - 1\,,\,A_0\,,\,K_1\,,\,A_1\,,\,\ldots\,,\,K_{\lceil m/2 \rceil - 1}\,,\,A_{\lceil m/2 \rceil - 1}$$

结点 v 中包含的信息为

$$m - \lceil m/2 \rceil\,,\,A_{\lceil m/2 \rceil}\,,\,K_{\lceil m/2 \rceil + 1}\,,\,A_{\lceil m/2 \rceil + 1}\,,\,\ldots\,,\,K_m\,,\,A_m$$

而关键字$K_{\lceil m/2 \rceil}$则插入到p的父结点中去。如果p的父结点不存在，则新建一个只包含关键字$K_{\lceil m/2 \rceil}$的结点；如果p的父结点关键字个数由于关键字$K_{\lceil m/2 \rceil}$的插入而超过m–1，则分裂过程继续下去，直到p的某个祖先结点g，在插入关键字后g的关键字个数不超过m–1。

例如在图 8-16（a）所示的 5 阶 B-树中，插入关键字 7 之后的结果如图（b）所示；若继续插入关键字 3，则插入关键字 3 后，其所在结点关键字个数大于 m–1=4，此时需要进行分裂，分裂结果如图（d）所示；若继续插入关键字 55、70、38，则插入过程如图（e）~（j）所示，在插入关键字 70 和 38 后，都会引起结点的分裂，尤其是在插入关键字 38 后，引起了多次结点分裂，直到产生了新的根结点之后分裂才停止，过程如图（h）~（j）所示。

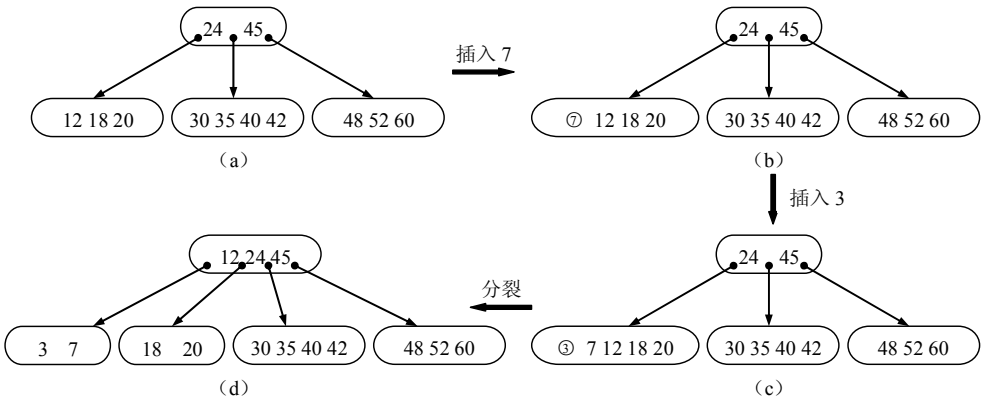


图 8-16 在 B-树中插入关键字

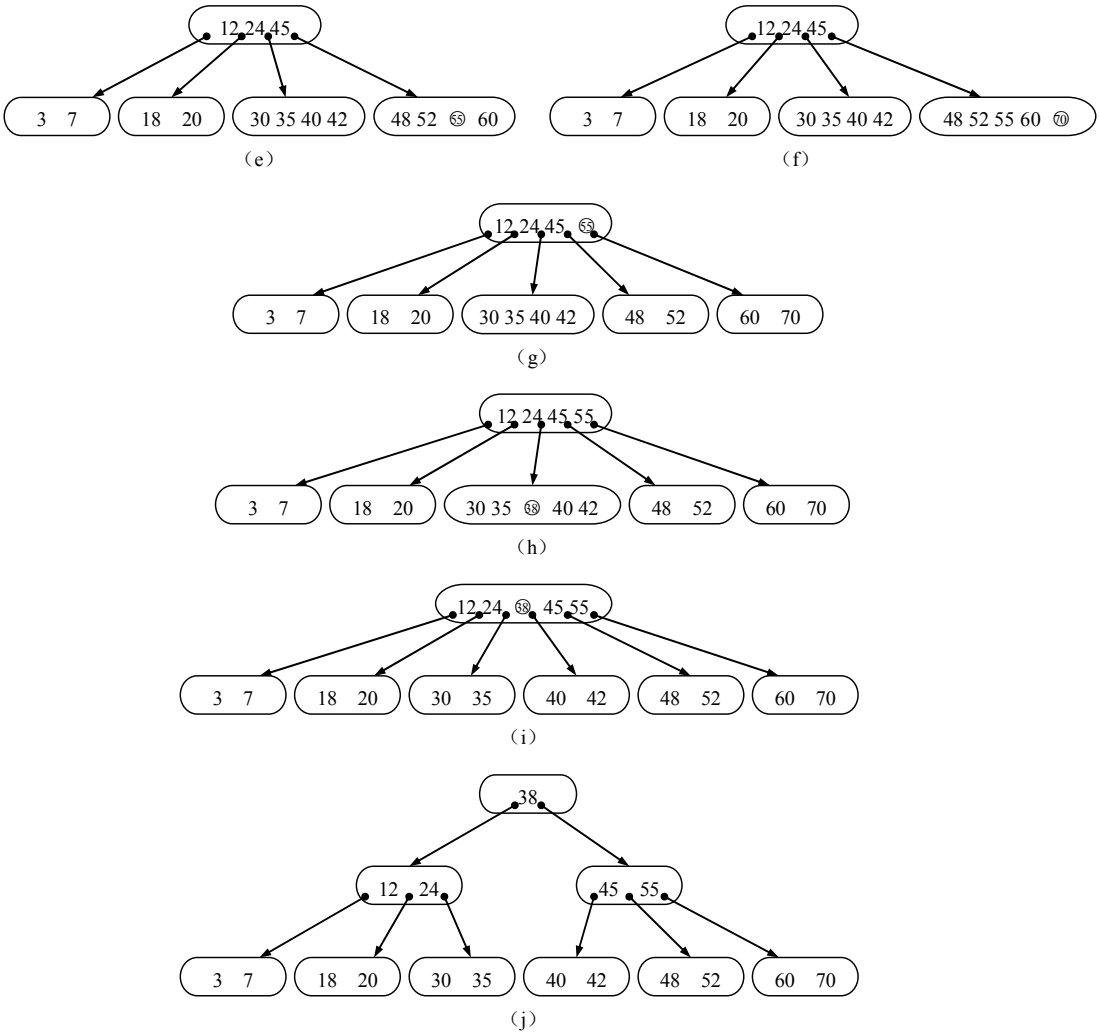


图 8-16 续

■ 关键字的删除操作

与插入关键字相反，若在 B-树上删除一个关键字，则首先应该找到该关键字所在的结点 u，并从中删除。

1. 若待删关键字所在结点并非最下层的非终端结点

假设待删关键字为K_i，此时，可以用A_i所指子树中的最小关键字X替代，然后删除关键字X即可。

2. 若该结点 u 为最下层的非终端结点，且其中关键字的数目不少于⌈m/2⌉，则直接删除之，否则在 u 中删除关键字后需要进行“合并”结点的操作。“合并”操作可以分成以下三种情况分别进行处理：

① 结点 u 的左兄弟 v 包含至少⌈m/2⌉个关键字

此时，如图 8-17 所示，可以从结点v中取出最大的关键字k_{max}，将u的父结点p中介于v和u之间的关键字k_{mid}替换为k_{max}，然后将k_{mid}插至u的最左侧。

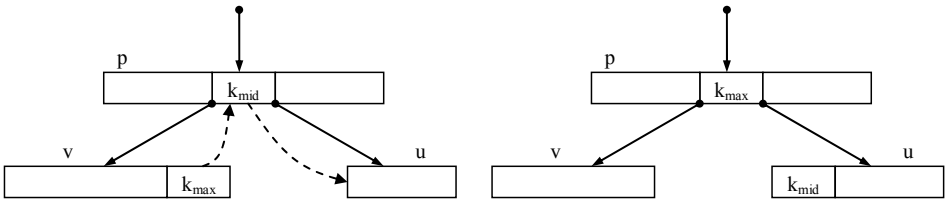


图 8-17 从左兄弟“借”关键字

② 结点 u 的右兄弟 v 包含至少⌈m/2⌉个关键字

此时，如图 8-18 所示，可以从结点v中取出最小的关键字k_{min}，将u的父结点p中介于v和u之间的关键字k_{mid}替换为k_{min}，然后将k_{mid}插至u的最右侧。

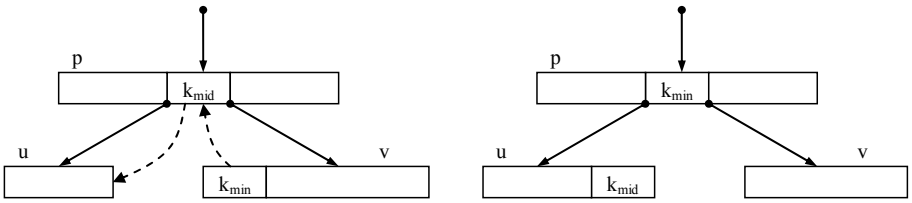


图 8-18 从右兄弟“借”关键字

③ 结点 u 没有一个兄弟包含至少⌈m/2⌉个关键字

此时，结点u和其相邻的兄弟结点中的关键字数目均等于⌈m/2⌉–1，但是结点u至少有一个兄弟结点。如图 8-19 所示，不妨设u有右兄弟v，这时可以取出父结点p中介于u、v之间的关键字k_{mid}，然后将k_{mid}与u、v中关键字合并为一个结点。

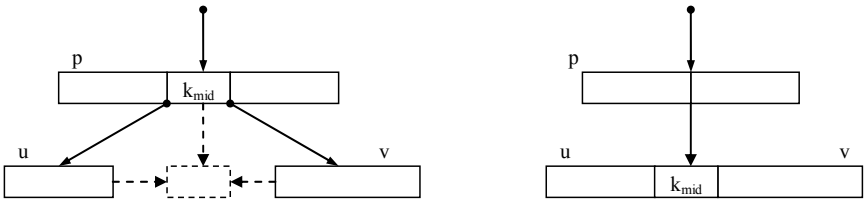
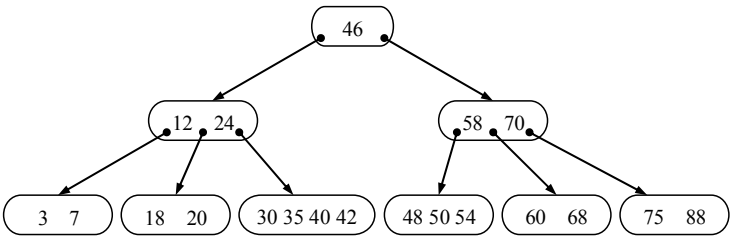


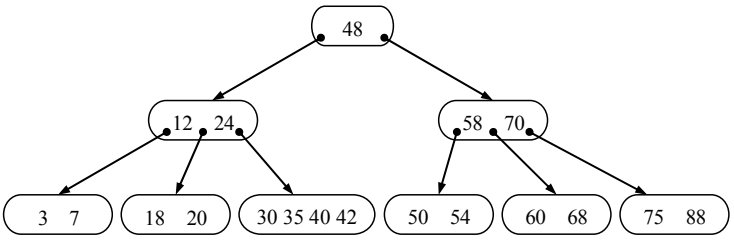
图 8-19 从父亲“借”关键字

综合以上三种情况，在“合并”结点之后，结点 u 的父结点中关键字不会增加，但是有可能会减少，如果父结点关键字数目小于⌈m/2⌉–1，则对父结点进行修复，该过程一直进行下去，直到 u 的某个祖先结点 g，g 的关键字数目不少于⌈m/2⌉–1，或直到根结点为止。

例如在图 8-20（a）所示的 5 阶 B-树中删除关键字 46，可以用关键字 48 替代，然后再删除关键字 48 即可，结果如图（b）所示；若继续删除关键字 20，因为 20 所在结点关键字数目达到下限⌈m/2⌉–1，而其右兄弟有多余的关键字，因此可以向其右兄弟“借”一个关键字 30，结果如图（c）所示；若在（c）基础上继续删除关键字 3，由于 3 所在结点的所有兄弟的关键字数目均达到下限，因此在删除 3 后，将剩余关键字 7 与父结点中关键字 12、以及其右兄弟结点中关键字一起组成一个新结点，如图（d）所示，由于父结点中减少一个关键字，导致父结点关键字数目小于⌈m/2⌉–1，因此再对父结点进行修复，如图（e）所示。

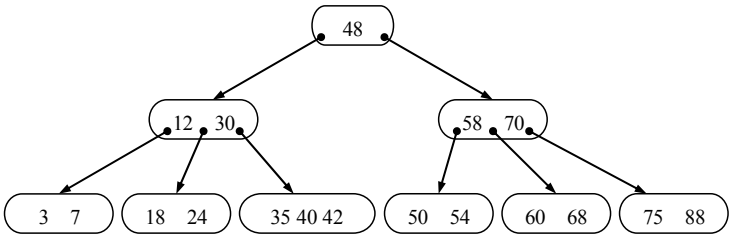


（a）

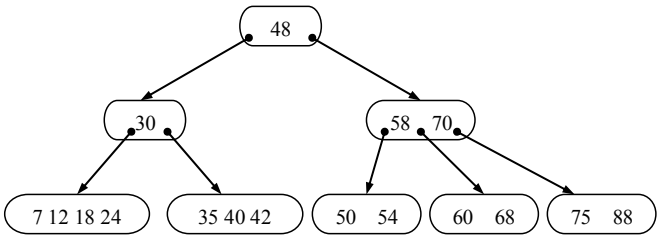


（b）

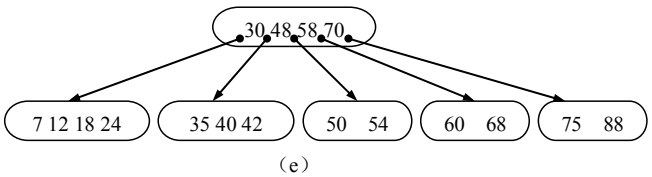
图 8-20 在 B-树中删除关键字



（c）



（d）



（e）

图 8-20 续

8.4 哈希

实现查找表的另一种有效数据结构是哈希表。虽然在哈希表里查找一个元素所需要的时间最坏的情况下可能和线性表一样，但是哈希表的平均查找效率是很高的。在一些合理的假定下，在哈希表中查找一个元素的期望时间为 O(1)。

8.4.1 哈希表

哈希表是通常的数组概念的推广。在一个普通的数组中，最有效的查找一个元素的方法是直接寻址，时间复杂度是 O(1)。直接寻址的基本思想是：如果存储空间允许,我们可以为每个关键字分配一个位置,每个元素的存储地址就是关键字对应的数组下标。在这种情况下应用直接寻找技术是很有效的。

图 8-21 说明了直接寻址方法。图 8-21 中数组 T[0..m-1]代表了直接寻址查找表，在数组中每个分量与关键字取值的所有可能一一对应，数组中位置 T[k]对应于关键字 k。如果在查找表中关键字 k 没有出现，则相应的位置 T[k]为空，如图中阴影部分所示。

然而使用直接寻址的问题是：当查找表中关键字取值的范围较小时直接寻址是一种非常好的方法，但是，如果关键字的可能取值范围 U 非常大,则要在一台典型的计算机上分配足够大的空闲内存来存放一个大小为|U|的表 T 也许是不切实际, 甚至是不可能的。此外, 实际要存储的关键字集合 K 相对 U 来说可能很小，则分配给 T 的大部分空间都要浪费掉。

当存放在查找表中的关键字集合K 比所有可能的关键字域U小很多时，哈希表要求比直接寻址表更少的存贮空间。具体地，存储空间要求下降到Θ(|K|)，这时在哈希表中查找一个元素仅需O(1)时间。在这里需要注意的是，这个是哈希表的平均执行时间，但对直接寻址而言是它在最坏情况下的执行时间。

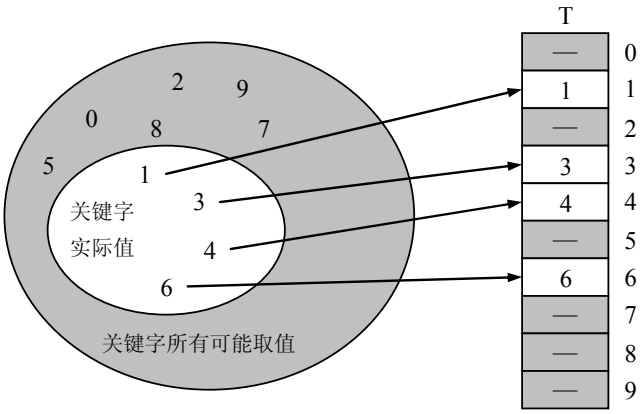


图 8-21 直接寻址表

在直接寻址方式下，一个元素关键字 k 被存放在位置 k。在哈希方式下，这个元素处于位置 h(k)；就是说，我们根据关键字 K 用哈希函数 h 计算出位置。函数 h 将关键字域 U 映射到哈希表 T[0..m - 1]的位置上:

$h : U \rightarrow \{0, 1, ..., m - 1\}$

我们可以说，一个具有关键字 k 的元素是被哈希到位置 h(k)上；我们也可以说，h(k) 是关键字 k 的散列（哈希）值。图 8-22 给出了形象的说明。采用哈希函数的关键作用是减少需要被处理的数组大小。和直接寻址相比，我们需要处理的数组大小为 m 而不是|K|，由此，储存空间的开销也相应减小。

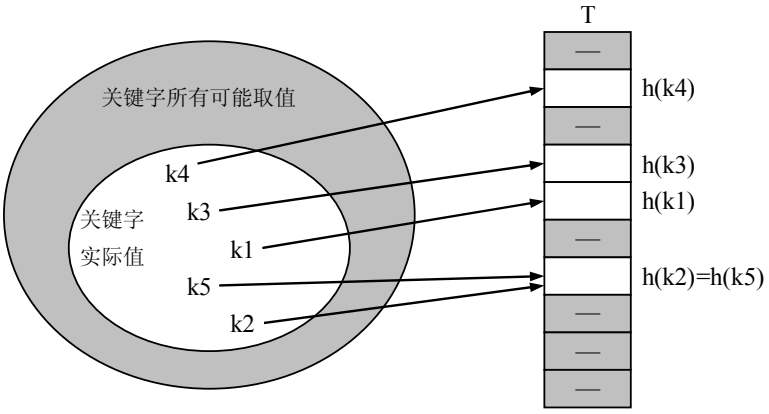


图 8-22 哈希表

这里存在一个问题：两个关键字可能哈希到相同的位置，我们称这种情况为冲突。此时这两个关键字称为同义词，例如图 8-22 中k2 与k5 即为同义词。目前，已经有一些有效的技术可以用来解决冲突问题。

当然，理想的情况是希望完全地避免冲突。要试图做到这一点我们可以选择一个适当的哈希函数 h。选择 h 的主导思想是使 h(k)的出现是“随机”的，从而避免冲突或至少使冲突的次数减到最小（当然, 哈希函数 h 必须是确定的，因为给一个输入 k 应该总指向同样输出 h(k)） 。然而，由于|U| > m，因此至少有两个关键字有相同的哈希值；因此完全地避免冲突是不可能的。因而，一方面我们可以使随机哈希函数尽量“随机”使冲突的次数减到最小，另一方面我们仍然需要解决可能出现冲突的办法。

在后面的两个小节中，我们介绍一些常见的哈希函数和解决冲突的方法。

8.4.2 哈希函数

许多哈希函数都假设关键字的值域为自然数集合 N={0, 1, 2, …}。因此，如果关键字不是自然数，我们可以找到一种方法把他们转化为自然数。例如，一个字符串在合适的基数计数法中就可以被转化为一个整数表示。例如，字符串"nt"首先可以用两个十进制整数 110 和 116 表示，因为在 ASCII 字符集中, n=110 且 t=116。那么，在以 128 为基数的计数法中, 字符串"nt"被表示成(112*128)+116=14452。通常情况下，在某一特定的应用中，构造一些方法把每个关键字转化成一个大的自然数是很简单的。在下面的讨论中，我们假定关键字都是自然数。

构造哈希函数的方法很多，然而要构造一个好的哈希函数却需要相当的知识和技巧。一个好的哈希函数应该（近似）满足简单一致分布的假设：即每个关键字等可能地散列到任一个地址中去。下面我们介绍几种常见的哈希函数构造方法

- **除留余数法**

在除留余数法中，为了构造哈希函数，我们通过取 k 除以 m 的余数来将关键字 k 映射到哈希表的 m 个不同地址的某一个中去。即，哈希函数为：

h(k) = k mod m

例如，如果哈希表的大小 m=12 并且关键字 k=100，那么 h(k)=4。因为它仅仅需要一个简单的除法操作，所以除留余数法的计算速度是相当快的。

当使用除留余数法时，我们通常要注意 m 的选择。如果 m 选择不当，将会产生大量冲突。例如，对于关键字集合{200, 205, 210, 215, … 690, 695, 700}，若选取 m=100，则每个关

键字的散列值至少和其他 4 个关键字的散列值产生冲突；而如果选取 m=101，则不会产生冲突。

并且，m也不应该为 2 的整数次幂，因为如果$m=2^p$，那么h(k)就是k的最低p位所代表的数字。除非我们事先知道关键字的分布使得k的最低p位的各种排列形式出现的可能性相同，否则在设计哈希函数时应当使得散列值依赖于k的每一位。我们通常把一个不太靠近 2 的整数次幂的素数作为m的值。

- **乘法散列**

在乘法散列法中，为了构造哈希函数，我们需要两步操作：第一，我们将关键字 k 乘以一个常量 A，它的取值范围为 0<A<1，并且提出 kA 的小数部分；接下来，我们将这个值乘以 m，并且取结果的底。简言之，哈希函数是：

h(k) = m * (kA - ⌊kA⌋)

乘法散列法的一个优势是对 m 的值没有什么特别的要求。对某个 m，一般指定它的值是 2 的幂(m=2p，其中 p 是整数)，因为这样我们很容易就能在电脑中实现哈希函数的计算。

尽管这个方法可以和任何的常量A一起工作，但是它和一些特定的值会工作的更好。最好的选择是依赖于被散列计算的数据的特性。Knuth[①]认为

$$A \approx \left(\sqrt{5}-1\right)/2 = 0.618033...$$

可能是较为理想的值。

8.4.3 冲突解决

解决冲突的方法主要有链地址法和开放地址法。

- **链地址法**

在链地址法中，我们把哈希到同一位置的所有元素都放在一个链接表中，如图 8-23 所示 。位置 j 中有一个指针，它指向由所有哈希到 j 的元素构成的链表的头，如果不存在这样的元素，则 j 中为 Null。

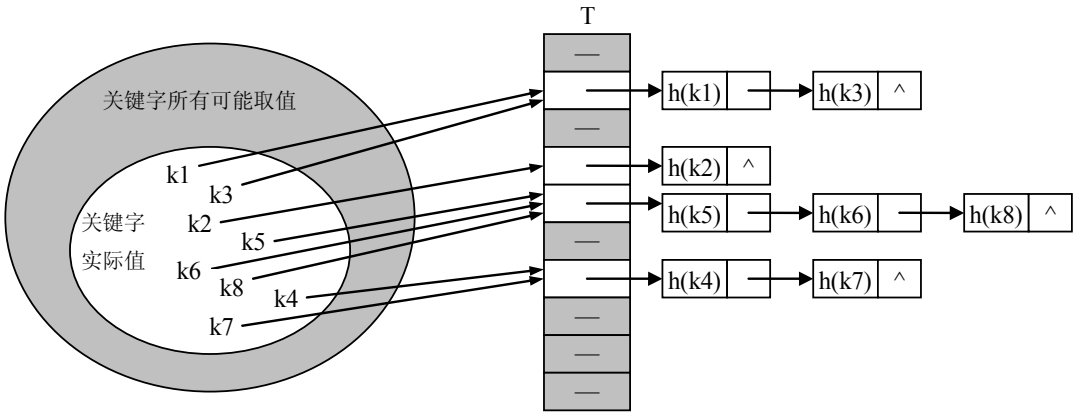


图 8-23 链地址法

在采用链地址法来解决冲突后，哈希表 T 上的添加、查找和删除操作就相对容易实现

①

了。如果假设元素 x 被插入前在表 T 中是不存在的，则插入操作在最坏情况下的运行时间为 O(1) 。如果需要判断 x 是否已经存在，则可以在插入之前执行一次查找操作。查找操作的最坏情况运行时间与 x 所在链表的长度成正比。如果链表是单链的，则要在表 T[h(key[x])]中去找 x，从而通过调整 x 的前趋的 next 指针来删除 x。在这种情况下，删除和查找的运行时间基本相同。

下面我们来具体分析一下，在采用链地址法时，各操作，尤其是查找操作的执行时间（若查找操作的执行时间为 Ts，则其他操作的时间为 O(Ts)）。

给定一个能存放 n 个元素的具有 m 个位置的哈希表 T，定义 T 的装载因子 α 为 n/m，即一个链中平均所存储的元素个数。我们的分析以 α 来表示（注意，α 可以小于、等于或大于 1）。

链地址法在最坏情况下性能很差：所有的 n 个关键字都哈希到同一个位置上，产生出一个长度为 n 的链表。这时，最坏情况下查找的时间为 Θ(n)，再加上计算哈希函数的时间，这不比用一个链表来存储所有元素好。

哈希方法的平均性能要依赖于所选的哈希函数。这里，假定任何元素哈希到m个位置的每一个的可能性是相同的，且与其他元素被哈希到什么位置是独立的。我们称此假设为简单一致哈希。在简单一致哈希的假设下，任何关键字k等可能地被哈希到m个位置的任一个之中。因而，一次不成功查找的平均时间就是对m个表中的某一个从头查找至尾的平均时间。这样一个表的平均长度即为装载因子α=n/m的大小。所以，一次不成功的查找平均要检查α个元素，总的时间（包括计算h(k)的时间）为Θ(1 + α)。同样，在简单一致哈希的假设下，我们也可以证明[①]，对用链地址法解决冲突的哈希表，平均情况下一次成功的查找也需要Θ(1 + α)时间。

上面的分析说明：如果哈希表的总地址数m至少与表中的元素数n成比例，我们有n = O(m)，从而α = n/m = O(m)/m = O(1)。因而，查找操作需要常数的时间。既然插入操作在最坏情况下需要O(1)时间，则所有的操作在平均情况下都可在O(1)时间内完成。

■ 开放地址法

解决冲突的另外一种方法是开放地址法。

在开放地址法中，所有的元素都存储在哈希表中，即哈希表中的每一个位置要么存储了某个元素，要么是Null。当采用这种方法时，要查找一个元素，我们需要通过一系列的探测才能找到或者判断元素不存在。因为在开放地址法中所有元素均存储在表中，因此装填因子α总是不大于 1。

(1) 线性探测

采用开放地址法解决冲突的最简单的一种形式是线性探测。假设$h'(k)$是一个哈希函数，那么线性探测法使用的哈希函数为：

$h(k, i) = (h'(k)+i) \bmod m \quad i=0, 1, 2, \ldots m-1$

具体地说，在执行插入操作时：首先计算 h(k, 0)，若 T[h(k, 0)]未被占用，则直接插入；否则尝试插入到 T[h(k, 1)]。要是 T[h(k, 1)]也被占用了，就继续尝试 T[h(k, 2)]。如此进行下去直到找到一个未被占用的存储单元。相应地，在执行查找操作时，首次对 T[h(k, 0)]的查找失败并不意味着在 T 中不包含 k，实际上我们未能还需要依次扫描 T[h(k, 1)]、T[h(k, 2)]...，直到发现 k（查找成功）或到达一个空单元（查找失败）。

例如，假设哈希表的长度m=11，哈希函数$h'(k)=k \bmod 11$。若依次插入关键字 25、48、60、36，其结果如图 8-24 所示。

①

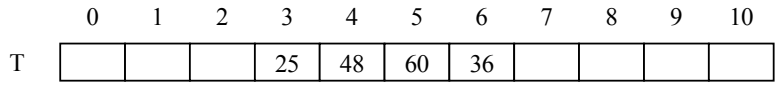


图 8-24 使用线性探测法解决冲突

由于 h (25, 0)=3，h(48, 0)=4，h(60,0)=5，因此关键字 25、48、60 分别占据 T[3]、T[4]和 T[5]的单元。如果再插入 36，由于 h (36, 0)=3，而 T[3]被 25 占据，所以继续计算 h (36, 1)=4，仍然冲突，因此继续计算 h (36, 2)=5，还是冲突，再次计算 h (36, 3)=6，此时不再冲突，最终将 36 放入 T[6]。

从上述例子可以看出，线性探测处理冲突容易产生元素“聚集”的现象，即在处理同义词的冲突时又导致了非同义词的冲突。线性探测的优点是只要哈希表不满，就一定能找到一个不冲突的哈希地址，然而，其缺点是当装填因子α较大时，元素聚集现象非常严重，这对执行元素查找操作极为不利。二次探测和双散列是克服这一缺点的有效方法。

(2) 二次探测

二次探测使用的哈希函数如下：

$h(k, i) = (h'(k)+i^2) \bmod m \quad i=0, 1, 2, \ldots m-1$

其中h'是一个辅助的哈希函数。采用二次探测可以很好的解决元素的聚集问题。这里充分利用了二次函数的特点，随着冲突次数的增加，其探测步长将快速增加，而不是像线性探测那样采用固定的步长 1。因此，一旦产生冲突，这一方法可以使带插入的元素迅速远离发生聚集的区域。

然而，这一方法也有不足。首先，尽管这一方法可以有效回避元素聚集的现象，但是还是会出现二次聚集的情况——元素虽然不会连续的聚集成片，但是会在多个间断的位置多次反弹。

其次，如果 m 选择不当，可能会造成出现虽然哈希表不满，但是由于循环反弹以至于无法插入的情况。

(3) 双散列

双散列也是一种有效解决元素聚集问题的方法。双哈希使用如下形式的哈希函数：

$h(k, i) = (h_1(k)+ih_2(k)) \bmod m \quad i=0, 1, 2, \ldots m-1$

其中$h_1(k)$和$h_2(k)$均是辅助哈希函数。在这里需要注意的是，对于任何k，$h_2(k)$都不能为 0，否则$h(k, 0)= h(k, 1)= h(k, 2)=...$，无法起到解决冲突的目的。

对于以上三种开放地址法，它们的平均查找性能也使用装填因子来表示。同样可以证明[①]，在简单一致哈希的假设下，一个具有装填因子为$\alpha=n/m<1$ 的开放地址哈希表，在一次不成功的查找中最多需要比较的元素个数的期望值不超过 $1/(1-\alpha)$。

①

第九章　排序

从第 9 章的内容容易看出，为了查找方便，通常希望计算机中的查找表是按关键字有序的，因为此时可以使用查找效率较高的折半查找。并且在实际的工程应用中经常会碰到排序的问题，因此学习和研究各种排序方法非常重要。本章介绍了排序的基本概念和几类重要的排序方法，从算法设计的角度看，这些算法体现了重要的程序设计思想和高超的程序设计技巧，为创造新方法提供了基础。

9.1 排序的基本概念

排序（sorting）的功能是将一个数据元素的任意序列，重新排列成一个按关键字有序的序列。其确切的定义为：

假设有n个数据元素的序列$\{R_1, R_2, \ldots, R_n\}$，其相应关键字的序列是$\{K_1, K_2, \ldots, K_n\}$，通过排序要求找出下标 $1, 2, \ldots, n$的一种排列$p_1, p_2, \ldots, p_n$，使得相应关键字满足如下的非递减（或非递增）关系

$$Kp_1 \le Kp_2 \le \ldots \le Kp_n$$

这样，就得到一个按关键字有序的纪录序列：$\{ Rp_1, Rp_2, \ldots, Rp_n \}$。

根据排序时待排序的数据元素数量的不同，使得排序过程中涉及的存储器不同，可以将排序方法分为两类。一类是整个排序过程在内存储器中进行，称为内部排序；另一类是由于待排序元素数量太大，以至于内存储器无法容纳全部数据，排序需要借助外部存储设备才能完成，这类排序称为外部排序。本章介绍的排序方法都属于内部排序。

上面所说的关键字K_i可以是元素R_i的主关键字，也可以是次关键字，甚至可以是若干数据项的组合。若K_i是主关键字，则任何一个元素的无序序列经排序后得到的结果是唯一的；若K_i是次关键字，则排序的结果不唯一。

如果在待排序的序列中存在多个具有相同关键字的元素。假设$K_i=K_j$（$1\le i\le n$，$1\le j\le n$，$i\neq j$），若在排序之前的序列中R_i在R_j之前，经过排序后得到的序列中R_i仍然在R_j之前，则称所用的排序方法是**稳定**的；否则，当相同关键字元素的前后关系在排序中发生变化，则称所用的排序方法是**不稳定**的。

无论是稳定的还是不稳定的排序方法，均能完成排序的功能。在某些场合可能对排序有稳定性的要求，此时就应当选择稳定的排序方法。例如，假设一组学生纪录已经按照学号有序，现在需要根据学生的成绩排序，当分数相同时要求学号小的学生在前，显然此时对分数进行排序就必须选择稳定的排序方法。

通常，在排序的过程中需要进行两种基本操作：⑴比较两个关键字的大小；⑵将一个关键字从一个位置移动到另一个位置。第一种操作对大多数排序方法来说都是必要的，而后一种操作可以通过改变记录的存储方式来予以避免。在本书中假设待排序的元素序列由一个 Object 数组给出，同时使用第三章中代码 3-3 定义的 Strategy 接口即可完成对两个数据元素的按关键字的比较。在具体的实现中，将所介绍的排序方法的实现都纳入到一个 Sorter 类中，将它们作为 Sorter 类的方法，如此在 Sorter 类中引入具体实现了 Strategy 接口的类的对象就可以完成按照元素关键字进行的比较操作。

内部排序的方法很多，但是很难说哪一种内部排序方法最好，每一种方法都有各自的优

缺点，适合于不同的环境下使用。如果按照排序过程中依据的原则对内部排序进行分类，则大致上可以分为插入排序、交换排序、选择排序、归并排序等排序方法。

9.2 插入类排序

插入排序的基本排序思想是：逐个考察每个待排序元素，将每一个新元素插入到前面已经排好序的序列中适当的位置上，使得新序列仍然是一个有序序列。

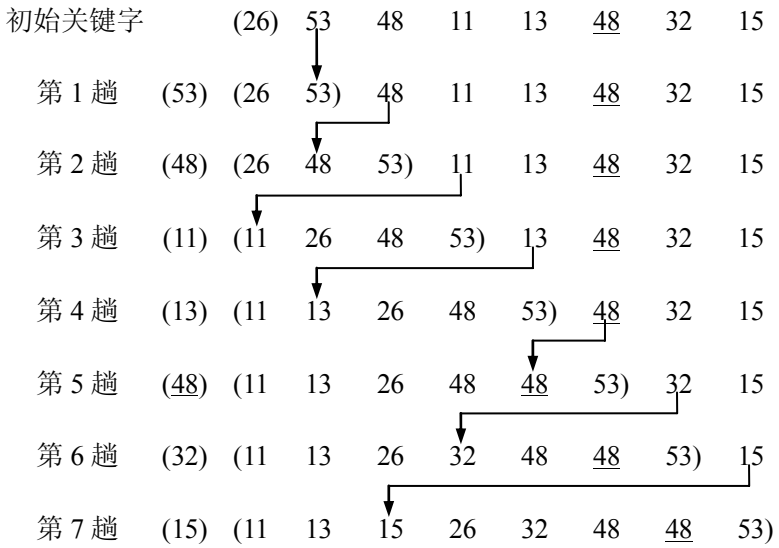
在这一类排序中主要介绍三种排序方法：直接插入排序、折半插入排序和希尔排序。

9.2.1 直接插入排序

直接插入排序是一种最简单的插入排序方法，它的基本思想是：仅有一个元素的序列总是有序的，因此，对 n 个记录的序列，可从第二个元素开始直到第 n 个元素，逐个向有序序列中执行插入操作，从而得到 n 个元素按关键字有序的序列。

一般来说，在含有 j-1 个元素的有序序列中插入一个元素的方法是：从第 j-1 个元素开始依次向前搜索应当插入的位置，并且在搜索插入位置的同时可以后移元素，这样当找到适当的插入位置时即可直接插入元素。

以关键字序列{ 26 , 53 , 48 , 11 , 13 , 48 , 32 , 15}为例，直接插入排序的过程如图 9-1 所示。



9-1 直接插入排序

算法 9-1 给出了直接插入排序法的具体实现。

算法 9-1 insertSort

输入：数据元素数组 r，数组 r 的待排序区间[low..high]

输出：数组 r 以关键字有序

代码：

```
public void insertSort(Object[] r, int low, int high){
    for (int i=low+1; i<=high; i++)
        if (strategy.compare(r[i],r[i-1])<0){      //小于时，需将 r[i]插入有序表
            Object temp = r[i];
```

```
                r[i] = r[i-1];
                int j=i-2;
                for (; j>=low&&strategy.compare(temp,r[j])<0; j--)
                    r[j+1] = r[j];              //记录后移
                r[j+1] = temp;                  //插入到正确位置
            }
        }
```

【效率分析】

空间效率：仅使用一个辅存单元。

时间效率：假设待排序的元素个数为 n，则向有序表中逐个插入记录的操作进行了 n-1 趟，每趟操作分为比较关键码和移动记录，而比较的次数和移动记录的次数取决于待排序列按关键码的初始排列。

⑴ 在最好情况下，即待排序序列已按关键字有序，每趟操作只需 1 次比较 0 次移动。此时有：

总比较次数 = n-1 次

总移动次数 = 0 次

⑵ 在最坏情况下，即待排序序列按关键字逆序排序，这时在第 j 趟操作中，为插入元素需要同前面的 j 个元素进行 j 次关键字比较，移动元素的次数为 j+1 次。此时有：

$$\text{总比较次数} = \sum_{j=1}^{n-1} j = n(n-1)/2 \text{ 次}$$

$$\text{总移动次数} = \sum_{j=1}^{n-1} (j+1) = (n+2)(n-1)/2 \text{ 次}$$

⑶ 平均情况下：即在第 j 趟操作中，插入记录大约需要同前面的 j/2 个元素进行关键字比较，移动记录的次数为 j/2+1 次。此时有：

总比较次数 $\approx n^2/4$ 次

总移动次数 $\approx n^2/4$ 次

由此，直接插入排序的时间复杂度为$O(n^2)$，并且是一个稳定的排序方法。

9.2.2 折半插入排序

从上一小节可见，直接插入排序算法简便、容易实现。当待排序元素的数量 n 很小时，这是一种较好的排序方法，但是通常待排序元素数量 n 很大，则不宜采用直接插入排序方法，此时需要对直接插入排序进行改进。

直接插入排序的基本操作是向有序序列中插入一个元素，插入位置的确定是通过对有序序列中元素按关键字逐个比较得到的。既然是在有序序列中确定插入位置，则可以不断二分有序序列来确定插入位置，即搜索插入位置的方法可以使用折半查找实现。

折半插入排序的实现如代码 9-2。

算法 9-2 binInsertSort

输入：数据元素数组 r，数组 r 的待排序区间[low..high]

输出：数组 r 以关键字有序

代码：

```
    public void binInsertSort(Object[] r, int low, int high){
```

```
        for (int i=low+1; i<=high; i++){
            Object temp = r[i];                    //保存待插入元素
            int hi = i-1;    int lo = low;         //设置初始区间
            while (lo<=hi){                        //折半确定插入位置
                int mid = (lo+hi)/2;
                if(strategy.compare(temp,r[mid])<0)
                    hi = mid - 1;
                else lo = mid + 1;
            }
            for (int j=i-1;j>hi;j--) r[j+1] = r[j]; //移动元素
            r[hi+1] = temp;                        //插入元素
        }//for
    }
```

从算法 9-2 容易看出，折半插入排序所需的辅助空间与直接插入排序相同，从时间上比较，折半插入排序仅减少了元素的比较次数，但是并没有减少元素的移动次数，因此折半插入排序的时间复杂度仍为$O(n^2)$。

9.2.3 希尔排序

希尔排序又称为“缩小增量排序”，它也是一种属于插入排序类的排序方法，是一种对直接插入排序的改进，但在时间效率上却有较大的改进。

从对直接插入排序的分析中知道，虽然直接插入排序的时间复杂度为$O(n^2)$，但是在待排序元素序列有序时，其时间复杂度可提高至O(n)。由此可知在待排序元素基本有序时，直接插入排序的效率可以大大提高。从另一方面看，由于直接插入排序方法简单，则在n值较小时效率也较高。希尔排序正是从这两点出发，对直接插入排序进行改进而得到的一种排序方法。

希尔排序的基本思想是：首先将待排序的元素分为多个子序列，使得每个子序列的元素个数相对较少，对各个子序列分别进行直接插入排序，待整个待排序序列“基本有序”后，再对所有元素进行一次直接插入排序。

根据上述排序思想，下面我们给出希尔排序的排序过程：

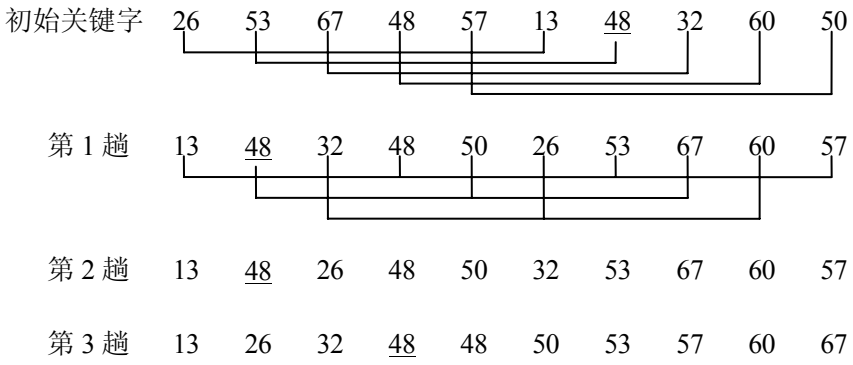
⑴ 选择一个步长序列t_1，t_2，…，t_k，其中$t_i>t_j$（i<j），$t_k=1$；

⑵ 按步长序列个数 k，对待排序元素序列进行 k 趟排序；

⑶ 每趟排序，根据对应的步长t_i，将待排序列分割成t_i个子序列，分别对各子序列进行直接插入排序。

当步长因子为 1 时，所有元素作为一个序列来处理，其长度为 n。

以关键字序列{ 26 , 53 , 67 , 48 , 57 , 13 , 48 , 32 , 60 , 50}为例，假设选择的步长序列为{5 , 3 , 1}，则希尔排序的过程如图 9-2 所示。因为步长序列长度为 3，因此对待排序序列一共需要进行 3 趟排序。首先，第一趟排序中将关键字序列分成 5 个子序列{26 , 13}，{53 , 48}，{67 , 32}，{48 , 60}，{57 , 50}，对它们分别进行直接插入排序，结果如图所示。然后，进行第二趟希尔排序，此时步长为 3，则将关键字序列分成 3 个子序列{13 , 48 , 53 , 57}，{48 , 50 , 67}，{32 , 26 , 60}，对它们进行直接插入排序后的结果如图所示。最后，对整个序列进行一趟直接插入排序，此时得到一个关键字有序的序列，希尔排序结束。



10-2 希尔排序

从图 9-2 所示希尔排序的例子中可以看到，在每趟排序过程中子序列的划分并不是简单的逐段划分，而是将间隔某个步长的元素组成一个子序列。如此，在对每个子序列进行简单插入排序时，关键字较小的元素就不是一步一步向前移动，而是按步长跳跃式向前移动，从而使得在进行最后一趟步长为 1 的插入排序时，整个序列已基本有序，此时，只需要作比较少的比较和移动即可完成排序。

希尔排序的算法如算法 9-3 所示。

算法 9-3 shellSort

输入：数据元素数组 r，数组 r 的待排序区间[low..high]，步长序列 delta

输出：数组 r 以关键字有序

代码：

```
public void shellSort(Object[] r, int low, int high, int[] delta){
    for (int k=0;k<delta.length;k++)
        shellInsert(r, low, high, delta[k]);  //一趟步长为 delta[k]的直接插入排序
}
private void shellInsert(Object[] r, int low, int high, int deltaK){
    for (int i=low+deltaK; i<=high; i++)
        if (strategy.compare(r[i],r[i-deltaK])<0){     //小于时，需将 r[i]  插入有序表
            Object temp = r[i];
            int j = i-deltaK;
            for(; j>=low&&strategy.compare(temp,r[j])<0; j=j-deltaK)
                r[j+deltaK] = r[j];                    //记录后移
            r[j+deltaK] = temp;                        //插入到正确位置
        }
}
```

通过前面的分析，从直观上我们可以预见希尔排序的效率会较直接插入排序要高，然而对希尔排序的时间复杂度分析是一个复杂的问题，因为希尔排序的时间复杂度与步长序列的选取密切相关，如何选择步长序列才能使得希尔排序的时间复杂度达到最佳，这还是一个有待解决的问题。但是，对于希尔排序的研究已经得出许多有趣的局部结论[①]。例如，当步长序列 $delta[k]=2^{t-k+1}-1$ 时，希尔排序的时间复杂度为 $O(n^{3/2})$，其中 t 为希尔排序的趟数，$1\le k\le t\le \lfloor \log(n+1) \rfloor$。实际的应用中，在选择步长序列时应当注意：应使步长序列中的步长值互质，并且最后一个步长值必须等于 1。

[①] 计算机程序设计艺术第三卷，第 5 章，P69-76。

9.3 交换类排序

交换类排序主要是通过两两比较待排元素的关键字，若发现与排序要求相逆，则“交换”之。在这类排序方法中最常见的是起泡排序和快速排序，其中快速排序是一种在实际应用中具有很好表现的算法。

9.3.1 起泡排序

起泡排序的思想非常简单。首先，将 n 个元素中的第一个和第二个进行比较，如果两个元素的位置为逆序，则交换两个元素的位置；进而比较第二个和第三个元素关键字，如此类推，直到比较第 n-1 个元素和第 n 个元素为止；上述过程描述了起泡排序的第一趟排序过程，在第一趟排序过程中，我们将关键字最大的元素通过交换操作放到了具有 n 个元素的序列的最一个位置上。然后进行第二趟排序，在第二趟排序过程中对元素序列的前 n-1 个元素进行相同操作，其结果是将关键字次大的元素通过交换放到第 n-1 个位置上。一般来说，第 i 趟排序是对元素序列的前 n-i+1 个元素进行排序，使得前 n-i+1 个元素中关键字最大的元素被放置到第 n-i+1 个位置上。排序共进行 n-1 趟，即可使得元素序列按关键字有序。

图 9-3 展示了一个起泡排序的实例。

初始关键字	26	53	48	11	13	48	32	15
第 1 趟	26	48	11	13	48	32	15	**53**
第 2 趟	26	11	13	48	32	15	**48**	**53**
第 3 趟	11	13	26	32	15	**48**	**48**	**53**
第 4 趟	11	13	26	15	**32**	**48**	**48**	**53**
第 5 趟	11	13	15	**26**	**32**	**48**	**48**	**53**
第 6 趟	11	13	**15**	**26**	**32**	**48**	**48**	**53**
第 7 趟	11	**13**	**15**	**26**	**32**	**48**	**48**	**53**

9-3 起泡排序

起泡排序的实现如算法 9-4 所示。

算法 9-4 bubbleSort

输入：数据元素数组 r，数组 r 的待排序区间[low..high]

输出：数组 r 以关键字有序

代码：

```
public void bubbleSort(Object[] r, int low, int high){
    int n = high - low + 1;
    for (int i=1;i<n;i++)
        for (int j=low;j<=high-i;j++)
            if (strategy.compare(r[j],r[j+1])>0)
            {
                Object temp = r[j];
                r[j] = r[j+1];
                r[j+1] = temp;
            }
```

}//end of bubbleSort

【效率分析】

空间效率：仅使用一个辅存单元。

时间效率：假设待排序的元素个数为 n，则总共要进行 n-1 趟排序，对 j 个元素的子序列进行一趟起泡排序需要进行 j-1 次关键字比较。由此，起泡排序的总比较次数为

$$\sum_{j=n}^{2}(j-1)=\frac{n(n-1)}{2}$$

因此，起泡排序的时间复杂度为$O(n^2)$。

9.3.2 快速排序

快速排序是将分治法运用到排序问题中的一个典型例子，快速排序的基本思想是：通过一个枢轴（pivot）元素将 n 个元素的序列分为左、右两个子序列 Ll 和 Lr，其中子序列 Ll 中的元素均比枢轴元素小，而子序列 Lr 中的元素均比枢轴元素大，然后对左、右子序列分别进行快速排序，在将左、右子序列排好序后，则整个序列有序，而对左右子序列的排序过程直到子序列中只包含一个元素时结束，此时左、右子序列由于只包含一个元素则自然有序。

用分治法的三个步骤来描述快速排序的过程如下：

4. 划分步骤：通过枢轴元素 x 将序列一分为二， 且左子序列的元素均小于 x，右子序列的元素均大于 x；
5. 治理步骤：递归的对左、右子序列排序；
6. 组合步骤：无

从上面快速排序算法的描述中我们看到，快速排序算法的实现依赖于按照枢轴元素 x 对待排序序列进行划分的过程。

对待排序序列进行划分的做法是：使用两个指针 low 和 high 分别指向待划分序列 r 的范围，取 low 所指元素为枢轴，即 pivot = r[low]。划分首先从 high 所指位置的元素起向前逐一搜索到第一个比 pivot 小的元素，并将其设置到 low 所指的位置；然后从 low 所指位置的元素起向后逐一搜索到第一个比 pivot 大的元素，并将其设置到 high 所指的位置；不断重复上述两步直到 low = high 为止，最后将 pivot 设置到 low 与 high 共同指向的位置。

使用上述划分方法即可将待排序序列按枢轴元素 pivot 分成两个子序列，当然 pivot 的选择不一定必须是 r[low]，而可以是 r[low..high]之间的任何数据元素。图 9-4 说明了一次划分的过程。

算法 9-5 实现了一次划分的过程。

算法 9-5 partition

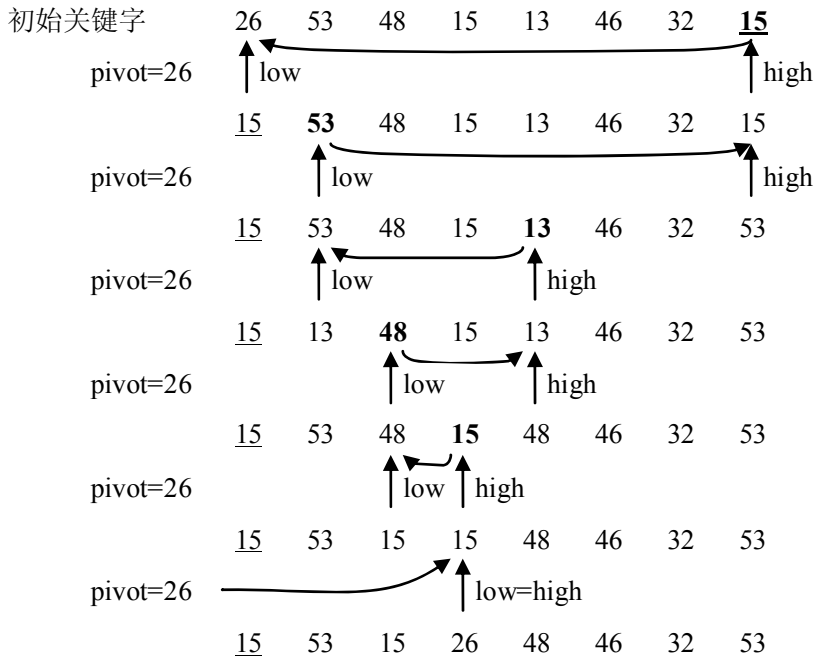
输入：数据元素数组 r，划分序列区间[low..high]

输出：将序列划分为两个子序列并返回枢轴元素的位置

代码：

```
private int partition(Object[] r, int low, int high){
    Object pivot = r[low];          //使用 r[low]作为枢轴元素
    while (low<high){               //从两端交替向内扫描
        while(low<high&&strategy.compare(r[high],pivot)>=0) high--;
        r[low] = r[high];           //将比 pivot 小的元素移向低端
        while(low<high&&strategy.compare(r[low],pivot)<=0) low++;
        r[high] = r[low];           //将比 pivot 大的元素移向高端
```

```
        }
        r[low] = pivot;                 //设置枢轴
        return low;                     //返回枢轴元素位置
    }
```



9-4 快速排序一次划分示例

在划分算法的基础上，快速排序算法的递归实现如算法 9-6 所示。

算法 9-6 quickSort

输入：数据元素数组 r，数组 r 的待排序区间[low..high]

输出：数组 r 以关键字有序

代码：

```
public void quickSort(Object[] r, int low, int high){
    if (low<high){
        int pa = partition(r,low,high);
        quickSort(r,low,pa-1);
        quickSort(r,pa+1,high);
    }
}
```

【效率分析】

时间效率：快速排序算法的运行时间依赖于划分是否平衡，即根据枢轴元素 pivot 将序列划分为两个子序列中的元素个数，而划分是否平衡又依赖于所使用的枢轴元素。下面我们在不同的情况下来分析快速排序的渐进时间复杂度。

快速排序的最坏情况是每次进行划分时，在所得到的两个子序列中有一个子序列为空。此时，算法的时间复杂度$T(n) = T_p(n) + T(n-1)$，其中$T_p(n)$是对具有n个元素的序列进行划分所需的时间，由以上划分算法的过程可以得到$T_p(n) = \Theta(n)$。由此，$T(n) = \Theta(n) + T(n-1) = \Theta(n^2)$。在快速排序过程中，如果总是选择r[low]作为枢轴元素，则在待排序序列本身已经有序或逆向有序时，快速排序的时间复杂度为$O(n^2)$，而在有序时插入排序的时间复杂度为O(n)。

快速排序的最好情况是在每次划分时，都将序列一分为二，正好在序列中间将序列分成长度相等的两个子序列。此时，算法的时间复杂度$T(n) = T_p(n) + 2T(n/2)$，由于$T_p(n) = \Theta(n)$，

所以T(n) = 2T(n/2) +Θ(n)，由master method知道T(n) = Θ(n log n)。

在平均情况下，快速排序的时间复杂度 T(n) = kn ln n，其中 k 为某个常数，经验证明，在所有同数量级的排序方法中，快速排序的常数因子 k 是最小的。因此就平均时间而言，快速排序被认为是目前最好的一种内部排序方法。

快速排序的平均性能最好，但是，若待排序序列初始时已按关键字有序或基本有序，则快速排序蜕化为起泡排序，其时间复杂度为$O(n^2)$。为改进之，可以采取随机选择枢轴元素pivot的方法，具体做法是，在待划分的序列中随机选择一个元素然后与r[low]交换，再将r[low]作为枢轴元素，作如此改进之后将极大改进快速排序在序列有序或基本有序时的性能，在待排序元素个数n较大时，其运行过程中出现最坏情况的可能性可以认为不存在。

空间效率：虽然从时间上看快速排序的效率优于前述算法，然而从空间上看，在前面讨论的算法中都只需要一个辅助空间，而快速排序需要一个堆栈来实现递归。若每次划分都将序列均匀分割为长度相近的两个子序列，则堆栈的最大深度为 log n，但是，在最坏的情况下，堆栈的最大深度为 n。

9.4 选择类排序

选择排序的基本思想是：每一趟从 n-i+1 (i=1,2,…,n)个元素中选取一个关键字最小的元素作为有序序列中第 i 个元素。本节在介绍简单选择排序的基础上，给出了对其进行改进的算法——树型选择排序和堆排序。

9.4.1 简单选择排序

简单选择排序的基本思想非常简单，即：第一趟，从 n 个元素中找出关键字最小的元素与第一个元素交换；第二趟，在从第二个元素开始的 n-1 个元素中再选出关键字最小的元素与第二个元素交换；如此，第 k 趟，则从第 k 个元素开始的 n-k+1 个元素中选出关键字最小的元素与第 k 个元素交换，直到整个序列按关键字有序。

图 9-5 给出了一个简单选择排序的示例。图中括号内的元素为当前候选元素，括号外的元素是已经排好序的元素。

算法 9-7 selectSort

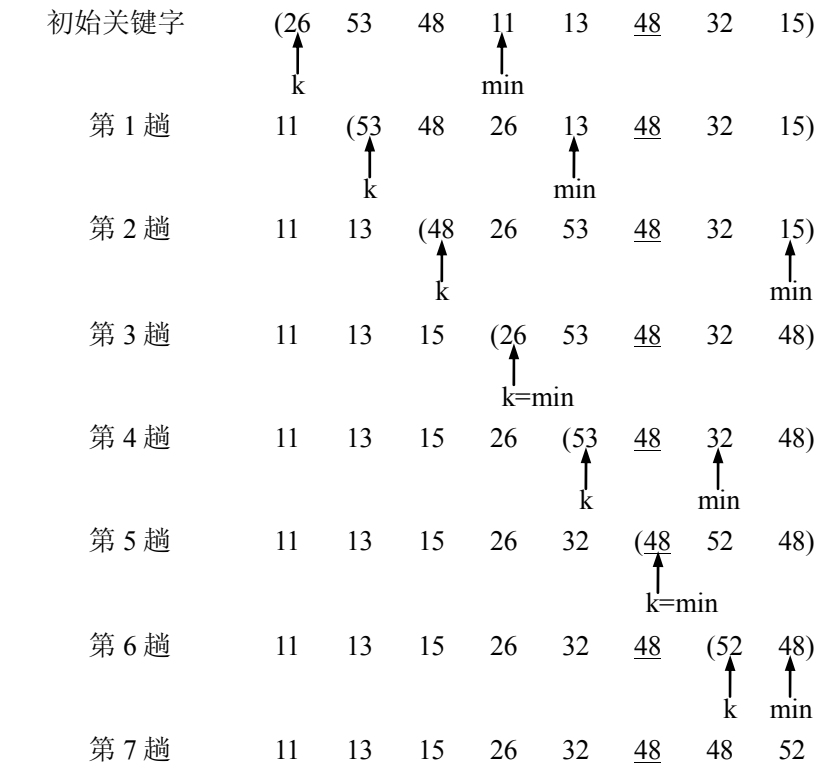
输入：数据元素数组 r，数组 r 的待排序区间[low..high]

输出：数组 r 以关键字有序

代码：

```
public void selectSort(Object[] r, int low, int high){
    for (int k=low; k<high-1; k++){          //作 n-1 趟选取
        int min = k;
        for (int i=min+1; i<=high; i++)     //选择关键字最小的元素
            if (strategy.compare(r[i],r[min])<0) min = i;
        if (k!=min){
            Object temp = r[k];              //关键字最小的元素与元素 r[k]交换
            r[k] = r[min];
            r[min] = temp;
        }//end of if
    }//end of    for(int k=0…
```

}//end of selectSort



9-5 简单选择排序

【效率分析】

空间效率：显然简单选择排序只需要一个辅助空间。

时间效率：在简单选择排序中，所需移动元素的次数较少，在待排序序列已经有序的情况下，简单选择排序不需要移动元素，在最坏的情况下，即待排序序列本身是逆序时，则移动元素的次数为 3(n-1)。然而无论简单选择排序过程中移动元素的次数是多少，在任何情况下，简单选择排序都需要进行n(n-1)/2 次比较操作，因此简单选择排序的时间复杂度为$O(n^2)$。

算法改进思想：从上述效率分析中可以看出，简单选择排序的主要操作是元素间的比较操作，因此改进简单选择排序应从减少元素比较次数出发。在简单选择排序中，首先从 n 个元素的序列中选择关键字最小的元素需要 n-1 次比较，在 n-1 个元素中选择关键字最小的元素需要 n-2 次比较……，在此过程中每次选择关键字最小的元素都没有利用以前比较操作得到的结果。欲降低比较操作的次数，则需要把以前比较的结果记录下来，由此得到一种改进的选择类排序算法，即树型选择排序。

9.4.2 树型选择排序

树型选择排序也称为锦标赛排序。其基本思想是：先把待排序的n个元素两两进行比较，取出较小者，若轮空则直接进入下一轮比较；然后在⌈n/2⌉个较小者中，采用同样的方法进行比较，再选出较小者；如此反复，直到选出关键字最小的元素为止。这个过程可以使用一颗具有n个结点的完全二叉树来表示，最终选出的关键字最小的元素就是这棵二叉树的根结点。例如图 9-6（a）给出了对 8 个元素的关键字{ 26 , 53 , 48 , 11 , 13 , 48 , 32 , 15}选出最小关键字元素的过程。

在输出关键字最小的元素后，为选　出次小关键字，可以将最小关键字元素所对应的叶

子结点的关键字设置为∞，然后从该叶子结点起逆行向上，将所经过的结点与其兄弟进行比较，修改从该叶子结点到根结点上各结点的值，则根结点的值即为次小关键字。例如，在图 9-6（a）的基础上输出最小关键字 11 后，输出次小关键字 13 的过程如图 9-6（b）所示。在图 9-6（b）的基础上输出最小关键字 13 后，输出次小关键字 15 的过程如图 9-6（c）所示。

重复上述过程，直到所有元素全部输出为止，则得到按关键字有序的序列。

【效率分析】

时间效率：在树型选择排序过程中为找到关键字最小的元素一共进行了 n-1 次比较，此后每次找出一个关键字所需要的比较次数等于完全二叉树的高度 h，而具有 n 个叶子结点的完全二叉树其高度为⌈log n⌉，由此可知除最小关键字外，每选择一个次小关键字需要进行⌈log n⌉次比较，因此树型选择排序的时间复杂度 T(n) = (n-1)⌈log n⌉ + (n-1) = O(n log n)。

空间效率：与简单选择排序相比，

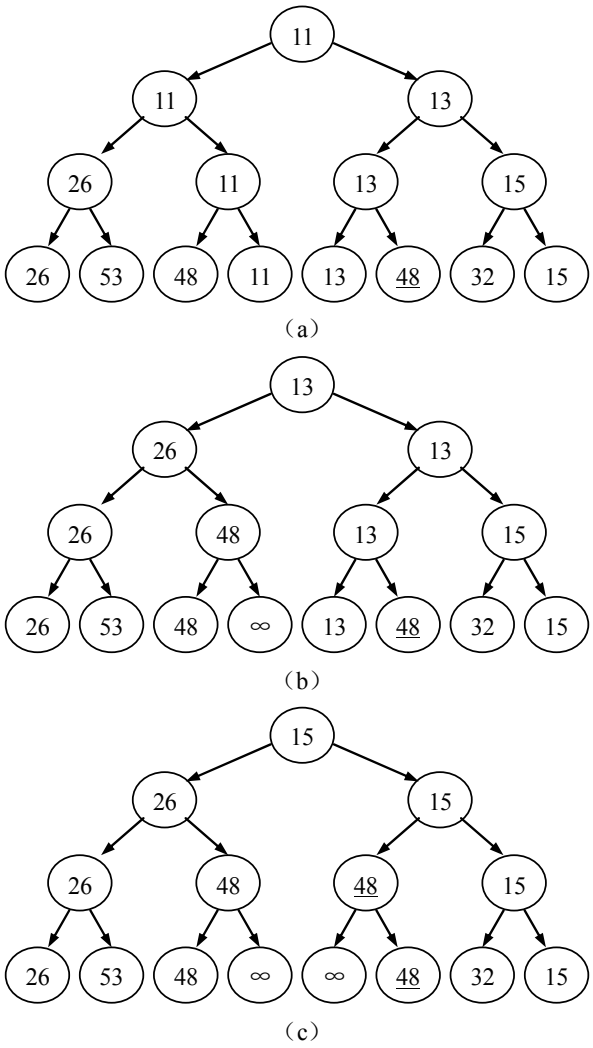


图 9-6 树型选择排序

虽然树型选择排序减小的时间复杂度，却使用了更多的辅助空间，在树型选择排序中共使用了 n-1 个而外的存储空间存放以前的比较结果。

算法改进思想：树型选择排序的缺点是使用了较多的辅助空间，以及和∞进行多余比较，为弥补树型选择排序的这些缺点，J.W.J.Williams 在 1964 年提出了进一步的改进方法，即堆排序。

9.4.3 堆排序

在介绍堆排序之前，首先介绍堆的概念。堆的定义为：n个元素的序列$\{k_1, k_2, \dots, k_n\}$，当且仅当满足下列关系时，称之为堆。

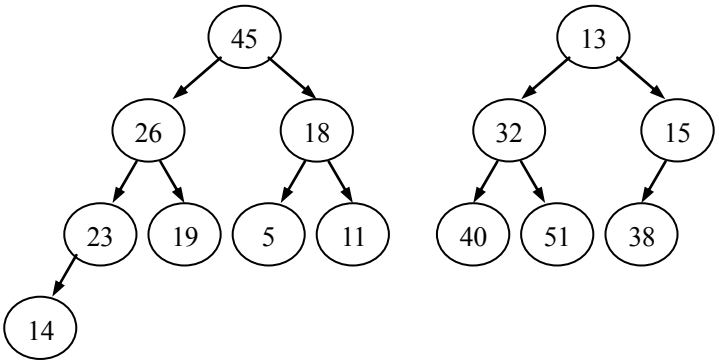
$$① \begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \quad 或② \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \quad 其中\ i=1, 2, \dots, \lfloor n/2 \rfloor$$

若满足条件①，则称为小顶堆，若满足条件②，则称为大顶堆。

如果将序列$\{k_1, k_2, \dots, k_n\}$对应为一维数组，且序列中元素的下标与数组中下标一致，即数组中下标为 0 的位置不存放数据元素，此时该序列可看成是一颗完全二叉树，则堆的定义说明，在对应的完全二叉树中非终端结点的值均不大于（或不小于）其左右孩子结点的值。

由此，若堆是大顶堆，则堆顶元素——完全二叉树的根——必为序列中n个元素的最大值；反之，若是小顶堆，则堆顶元素必为序列中n个元素的最小值。

例如图 9-7 显示了两个堆，其对应的元素序列分别为{45 , 26 , 18 , 23 , 19 , 5 , 11 , 14}、{13 , 32 , 15 , 40 , 51 , 38}。其中（a）是一个大顶堆，（b）是一个小顶堆。



（a）　　（b）

9-7 堆

设有 n 个元素，欲将其按关键字排序。可以首先将这 n 个元素按关键字建成堆，将堆顶元素输出，得到 n 个元素中关键字最大（或最小）的元素。然后，再将剩下的 n-1 个元素重新建成堆，再输出堆顶元素，得到 n 个元素中关键字次大（或次小）的元素。如此反复执行，直到最后只剩一个元素，则可以得到一个有序序列，这个排序过程称之为堆排序。

从对排序的过程中可以看到，在实现对排序时需要解决两个问题：

1. 如何将 n 个元素的序列按关键字建成堆；

2. 输出堆顶元素后，怎样调整剩余 n-1 个元素，使其按关键字成为一个新堆。

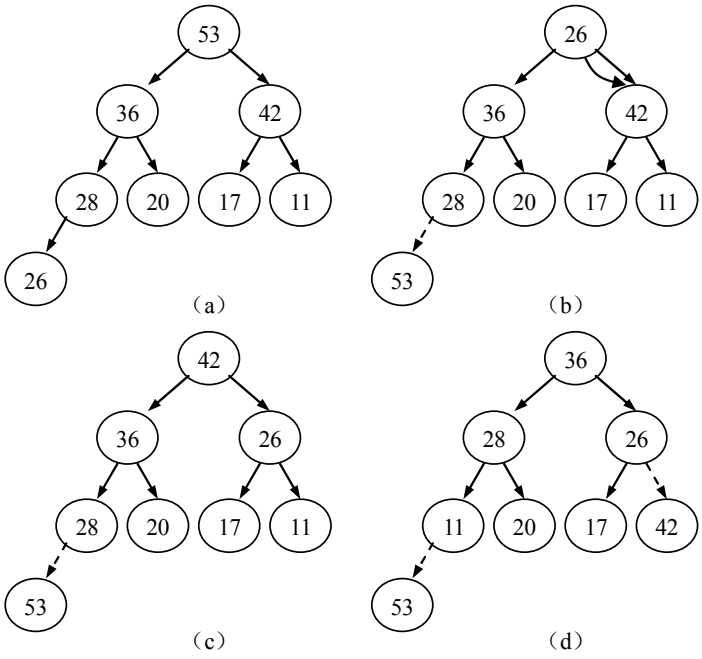
我们首先第二个问题，即输出堆顶元素后，对剩余元素重新建成堆的调整过程。

设有一个具有 m 个元素的堆，输出堆顶元素后，剩下 m-1 个元素。具体的调整方法是：首先，将堆底元素（最后一个元素）送入堆顶，此时堆被破坏，其原因仅是根结点不满足堆的性质，而根结点的左右子树仍是堆。然后，将根结点与左、右子女中较大（或较小）的进行交换。若与左孩子交换，则左子树堆被破坏，且仅左子树的根结点不满足堆的性质；若与右孩子交换，则右子树堆被破坏，且仅右子树的根结点不满足堆的性质。继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，则堆被重建。我们称这个自根结点到叶子结点的调整过程为筛选。

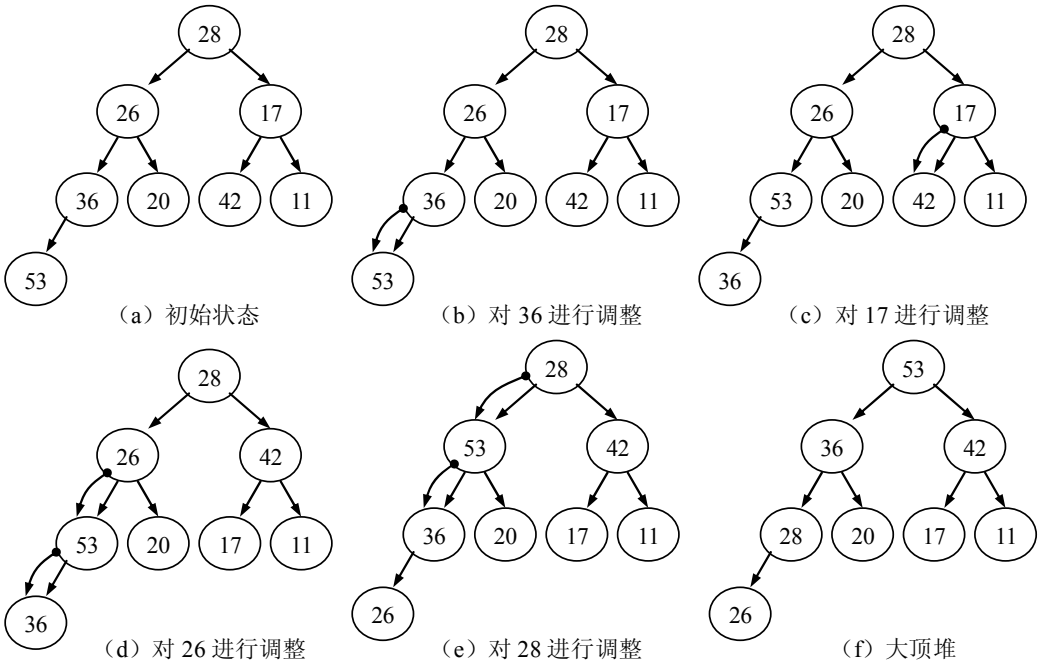
例如图 9-8（a）为一个大顶堆，在输出堆顶元素 53 之后，将堆底元素 26 送入堆顶，如图 9-8（b）所示；然后将 26 与 36、42 中大的元素交换，交换后，以 26 为根的子树已是一个堆；此时筛选结束，得到一个新堆，如图 9-8（c）所示。如果继续输出堆顶元素 42，然后重建堆，则结果如图 9-8（d）所示。

由此，如果我们能够建立一个堆，那么排序的过程就是不断输出堆顶并进行筛选的过程。现在的关键问题是如何由一个元素的初始序列构造一个堆，实际上建堆的方法是逐层向上对每个非终端结点进行一次筛选即可。

例如，对于待排序的初始序列{28 , 26 , 17 , 36 , 20 , 42 , 11 , 53}，初始建堆的过程如图 9-9 所示。（a）是由初始序列得到的完全二叉树；初始建堆首的过程，以按层从下到上的第一个非叶子结点开始，即从 36 开始，对 36 进行调整，过程如图（b）所示，调整结果如图（c）所示；然后对下一个非叶子结点 17 进行调整，调整过程如图（c），结果如图（d）所示；继续上述过程直到根结点 28 为止，对 28 进行调整后，即得到一个大顶堆，结果如图（f）所示。



9-8 筛选过程



9-9 初始建堆过程

从上面的分析中我们可以看到，无论是初始建堆还是进行排序，都需要完成以某个元素为根的调整操作。算法 9-8 给出了调整操作的实现。

算法 9-8 heapAdjust

输入：数据元素数组 r，数组 r 的待调整区间[low..high]

输出：调整 r[low..high]使之成为大顶堆

代码：

```
//已知 r[low..high]中除 r[low]之外，其余元素均满足堆的定义
private void heapAdjust(Object[] r, int low, int high){
    Object temp = r[low];
    for (int j=2*low; j<=high; j=j*2){        //沿关键之较大的元素向下进行筛选
```

```
            //j 指向关键之较大的元素
            if (j<high&&strategy.compare(r[j],r[j+1])<0) j++;
            //若 temp 比其孩子都大，则插入到 low 所指位置
            if (strategy.compare(temp,r[j])>=0) break;
            r[low] = r[j];   low = j;   //向下筛选
        }
        r[low] = temp;
    }
```

在算法 9-8 的基础上，则可以实现初始化建堆和排序的过程。

算法 9-8 heapSort

输入：数据元素数组 r

输出：对 r[1..length-1]排序

代码：

```
    public void heapSort(Object[] r){
        int n = r.length - 1;
        for (int i=n/2; i>=1; i--)        //初始化建堆
            heapAdjust(r,i,n);
        for (int i=n; i>1; i--){          //不断输出堆顶元素并调整 r[1..i-1]为新堆
            Object temp = r[1];           //交换堆顶与堆底元素
            r[1] = r[i];
            r[i] = temp;
            heapAdjust(r,1,i-1);          //调整
        }
    }
```

注：为了代码的易读性，在算法 9-8 中对数组 r 进行排序时，排序的范围是[1..length-1]，这一点和前面的算法是不一样的，前面的算法其排序范围是由参数指定的。当然堆排序也可以对指定范围内的元素进行排序，只不过在对下标进行操作之前都必须进行相应的处理，读者可自行设计实现指定范围的堆排序算法。

【效率分析】

空间效率：显然堆排序只需要一个辅助空间。

时间效率：首先，对于深度为k的堆，heapAdjust算法中所需执行的比较次数至多为 2k 次。则在初始建堆的过程中，对于具有n个元素、深度为h的堆而言，由于在i层上最多有 2^i个结点，以这些结点为根的二叉树深度最大为h-i，那么⌊n/2⌋次调用heapAdjust时总共进行的关键字比较次数T_{init}为：

$$T_{init}=\sum_{i=h-1}^{0}2^i\bullet 2(h-i)=O\left(\sum_{j=1}^{h}2^{h-j}\bullet j\right)=O\left(n\sum_{j=1}^{h}\frac{j}{2^j}\right)=O(n)\quad 因为\sum_{i=0}^{\infty}\frac{i}{2^i}=2$$

即，初始化需要执行的比较操作的次数为 O(n)。

其次，在排序过程中，每输出一次堆顶元素需要进行一次调整，而每次调整所需的比较次数为 O(log n)，因此 n 次输出总共需要的比较次数为 O(n log n)。

由此，堆排序在任何情况下，其时间复杂度为 O(n log n)。这相对于快速排序而言是堆排序的最大优点。

堆排序在元素较少时由于消耗较多时间在初始建堆上，因此不值得提倡，然而当元素较多时还是很有效的排序算法。

9.5 归并排序

归并排序是另一类不同的排序方法，这种方法是运用分治法解决问题的典型范例。

归并排序的基本思想是基于合并操作，即合并两个已经有序的序列是容易的，不论这两个序列是顺序存储还是链式存储，合并操作都可以在 O(m+n)时间内完成（假设两个有序表的长度分别为 m 和 n）。为此，由分治法的一般设计步骤得到归并排序的过程为：

1. 划分：将待排序的序列划分为大小相等（或大致相等）的两个子序列；
2. 治理：当子序列的规模大于 1 时，递归排序子序列，如果子序列规模为 1 则成为有序序列；
3. 组合：将两个有序的子序列合并为一个有序序列。

图 9-10 显示了归并算法的执行过程。假设待排序序列为{4, 8, 9, 5, 2, 1, 4, 6}，如图所示，归并排序导致了一系列递归的调用，而这一系列调用过程可以由一个二叉树来表示。树中每个结点由两个序列组成，上端为该结点所表示的递归调用的输入，而下端为相应递归调用的输出。树中的边用表示递归调用方向的两条边取代，边上的序号表示各个递归调用发生的次序。

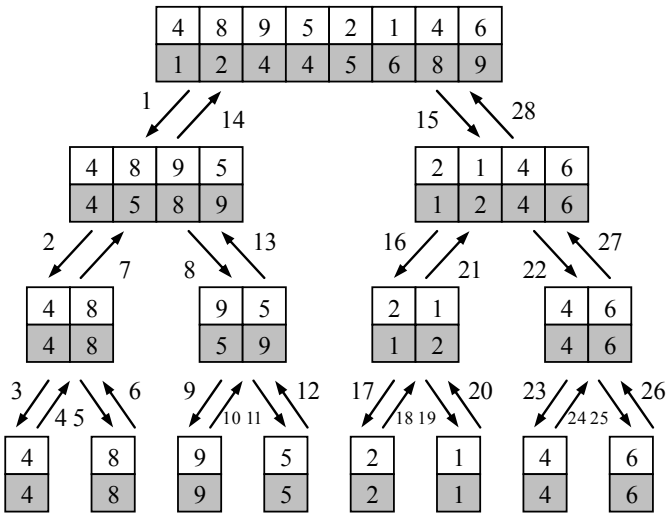


图 9-10 归并排序

归并排序中一个核心的操作是将一个序列中前后两个相邻的子序列合并为一个有序序列，算法 9-9 给出了合并操作的实现。

算法 9-9 merge

输入：数据元素数组 a，a 待合并的两个有序区间[p..q]以及[q+1..r]

输出：将两个有序区间合并为一个有序区间

代码：

```
private void merge(Object[] a, int p, int q, int r){
    Object[] b = new Object[r-p+1];
    int s = p;
    int t = q+1;
    int k = 0;
    while (s<=q&&t<=r)
        if (strategy.compare(a[s],a[t])<0)
            b[k++] = a[s++];
        else
```

```
                b[k++] = a[t++];
            while (s<=q) b[k++] = a[s++];
            while (t<=r) b[k++] = a[t++];
            for (int i=0; i<b.length; i++)
                a[p+i] = b[i];
        }
```

算法 merge 的时间复杂度为Θ(n)。因为假设待合并的两个子序列总长为 n，则这 n 个元素在从数组 a 移动到 b 的过程中，每个元素移动一次，而每次元素移动最多只需要一次比较；最后从数组 b 移回 a 也只需要 n 次移动操作即可，因此，算法 merge 的时间复杂度为Θ(n)。

在算法 merge 的基础上，即可实现归并排序的算法了。

算法 9-10 mergeSort

输入：数据元素数组 r

输出：对 r[low..high]排序

代码：

```
    public void mergeSort(Object[] r, int low, int high){
        if (low<high){
            mergeSort(r,low,(high+low)/2);
            mergeSort(r,(high+low)/2+1,high);
            merge(r,low,(high+low)/2,high);
        }
    }
```

【效率分析】

空间效率：在归并排序中，为了将子序列合并需要使用额外的存储空间，这个辅助存储空间的最大值不超过 n，因此归并算法的空间复杂度为Θ(n)。

时间效率：归并算法是一个典型的分治算法，因此，它的时间复杂度可以用 Master Method 来求解。通过对算法的分析我们写出算法时间复杂度的递推关系式：

$T(n) = 2T(n/2) + \Theta(n)$

该递推式满足 Master Method 的第二种情况，因此 $T(n) = O(n \log n)$。

与快速排序和对排序相比，归并排序的优点是它是一种稳定的排序方法。上述算法实现是归并排序的递归形式，这种形式简洁易懂，然而实用性较差，归并排序还可以按照自底向上的方式给出其他实现，读者可以自行设计完成。

9.6 基于比较的排序的对比

上面介绍的插入排序、交换排序、选择排序、归并排序等排序方法，都有一个共同的特点，那就是它们都是通过比较元素的大小来确定元素之间的相对位置的，即上述排序方法都是基于比较的排序方法。下面，我们就基于比较的排序方法进行一个对比和总结。

我们主要从算法的平均时间复杂度、最坏时间复杂度、空间复杂度以及排序的稳定性等方面，对各中排序方法加以比较。如表 9-1 所示。

表 9-1 各种排序方法的性能比较

排序方法	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
起泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定

快速排序	O(n log n)	$O(n^2)$	O(log n)	不稳定
简单选择排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
堆排序	O(n log n)	O(n log n)	O(1)	不稳定
归并排序	O(n log n)	O(n log n)	O(n)	稳定

从时间性能上看，快速排序是所有排序算法中实际性能最好的，然而快速排序在最坏情况下的时间性能不如堆排序和归并排序。这一点可以通过对快速排序进行改进来避免，一种通过随机选择枢轴元素的随机快速排序，可以使得出现最坏情况出现的几率非常小，在实际的运用中可以认为不存在。在堆排序和归并排序的比较中，当 n 较大时，归并排序所需时间较少，然而它需要较多的辅助存储空间。

从方法稳定性上来看，大多数时间复杂度为$O(n^2)$的排序均是稳定的排序方法，除简单选择排序之外。而多数时间性能较好的排序方法，例如快速排序、堆排序、希尔排序都是不稳定的。一般来说，排序过程中的比较是在相邻的两个元素之间进行的排序方法是稳定的。并且，排序方法的稳定性是由方法本身决定的，对于不稳定的排序方法而言，不管其描述形式如何，总能找到一种不稳定的实例。

综上所述，上面讨论的所有排序方法中，没有哪一个是绝对最优的，在实际的使用过程中，应当根据不同情况选择适当的排序方法。

在本节最后需要讨论的一个问题是：基于比较的排序方法，其可能达到的最快速度是什么，即排序方法的时间复杂度下界。

所有基于比较的排序方法的过程，均可以用一棵类似于图 9-11 所示的判定树来描述。

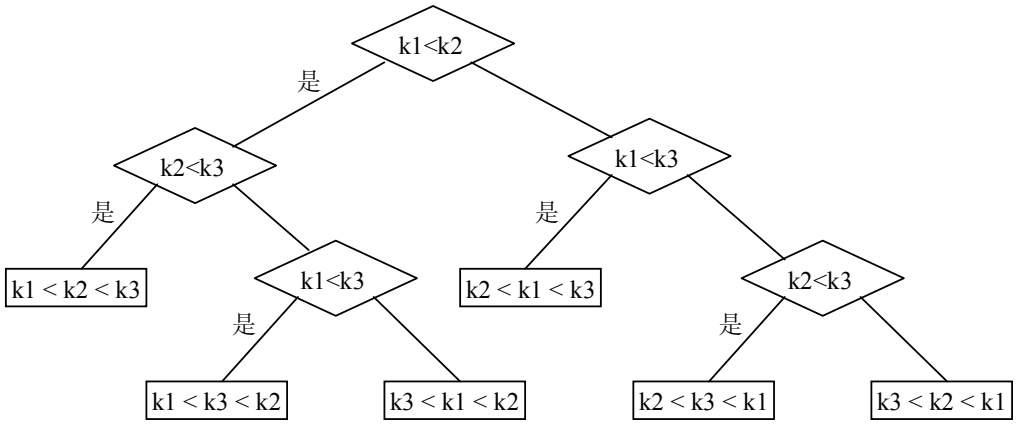


图 9-11 描述排序过程的判定树

图 9-11 所示的判定树表示 3 个关键字分别为 k1、k2 和 k3 的元素进行直接插入排序的过程，树中每个非终端结点表示两个元素间的一次比较，其左右子树分别表示这次比较的结果。假设 k1 ≠ k2 ≠ k3，则 3 个元素之间只可能有如下 6 种大小关系：(1) k1 < k2 < k3；(2) k1 < k3 < k2；(3) k2 < k1 < k3；(4) k2 < k3 < k1；(5) k3 < k1 < k2；(6) k3 < k2 < k1，也就是说，在经过排序之后只可能得到以下 6 种结果：(1){ k1, k2, k3}；(2){ k1, k3, k2}；(3){ k2, k1, k3}；(4){ k2, k3, k1}；(5){ k3, k1, k2}；(6){ k3, k2, k1}，而图 9-11 中判定树的 6 个叶子结点恰好表示了这 6 种排序结果。判定树上进行的每一次比较都是必要的，因此，这个判定树足以描述基于比较操作的排序过程。并且，对于每个初始序列经过排序达到有序所需比较的次数，恰为从根结点到和该序列对应的叶子结点的路径长度。由图 9-11 可以看出，对 3 个关键字进行排序，则至少需要进行 3 次比较。

推广至一般的情况，对 n 个元素进行排序至少需要多少次比较，这个问题等价于：含有 n!个叶子结点的二叉树的最小高度是多少。由于 n 个元素的序列可能出现的排序结果有 n!个，则描述 n 个元素排序过程的判定树必有 n!个叶子结点。

引理 9-1 设 T 是一棵有 n!个叶子结点的二叉树，则 T 的高度至少是 Ω(n long n)。

证明：设l是T中叶子结点数，并设h是它的高度。通过观察知道，当所有叶子结点出现在h层时，叶子结点达到最大值，此时，h层上结点最多可以有 2^h个。因为l=n!，我们有

$$n! = l \le 2^h$$

因此，$h \ge \log n! = \sum_{j=1}^{n} \log j \ge n\log n - n\log e + \log e \ge n\log n - 1.5n = \Omega(n\log n)$。

由此，我们得到以下结论：任何一个基于比较操作的排序方法，在最坏情况下所需要进行的比较次数至少为 n log n 次，即算法的时间复杂度下界为 Ω(n long n)。

9.7 在线性时间内排序

通过上一小节，我们知道通过比较确定两个元素之间相对位置的比较排序算法的时间复杂性下界为 O(n log n)，然而当排序序列满足某种特定条件时，我们可以突破这个时间下界，在线性时间内就可以完成排序。

9.7.1 计数排序

计数排序是一个非基于比较的线性时间排序算法。它对输入的数据有附加的限制条件：

1. 输入的线性表的元素属于有限偏序集 S；
2. 设输入的线性表的长度为 n，|S|=k（表示集合 S 中元素的总数目为 k），则 k=O(n)。

在这两个条件下，计数排序的复杂性为 O(n)。

计数排序算法的基本思想是对于给定的输入序列中的每一个元素 x，确定该序列中值小于 x 的元素的个数。一旦确定了这一信息，就可以将 x 直接存放到最终的输出序列的正确位置上。例如，如果输入序列中只有 9 个元素的值小于 x 的值，则 x 可以直接存放在输出序列的第 10 个位置上。当然，如果有多个元素具有相同的值时，我们不能将这些相同的元素放在输出序列的同一个位置上，因此还需对上述方案作适当的修改。

假设输入的序列L的长度为n，$L=\{L_0, L_1, \ldots, L_{n-1}\}$；线性表的元素属于有限偏序集S，|S|=k且k=O(n)，$S=\{S_0, S_1, \ldots S_{k-1}\}$；则计数排序算法可以描述如下：

1. 扫描整个集合S，对每一个$S_i \in S$，找到在序列L中小于等于S_i的元素的个数$C(S_i)$；
2. 扫描整个序列L，对L中的每一个元素L_i，将L_i放在输出线性表的第$C(L_i)$个位置上，并将$C(L_i)$减 1。

在实现计数排序的过程中，我们需要两个辅助数组：B[0..n-1]存放排序结果，C[0..k-1]作为临时数组。图 9-12 说明了计数排序的过程。

从图 9-12 所示的计数排序过程中可以看到，整个排序过程只需要对数组 A 进行两次扫描即可完成整个排序过程。首先，在从左到右的扫描数组 A 的过程中，对 S 中所有元素（即{0, 1, 2, 3, 4, 5}）出现的次数进行计数，将统计结果存放在临时数组 C 中，C 中每个分量 C[i]的取值即为 i 出现的次数。然后，使用第一步的统计结果，计算各个元素的输出位置。最后，从右到左扫描 A，对其中每个元素根据 C 中所指位置输出，并将对应输出位置大小减 1。

通过以上对计数排序过程的分析，我们知道计数排序的时间复杂度 T(n) = Θ(n + k)，由于 k=O(n)，因此，T(n) = Θ(n)。并且计数排序是一种稳定的排序方法。

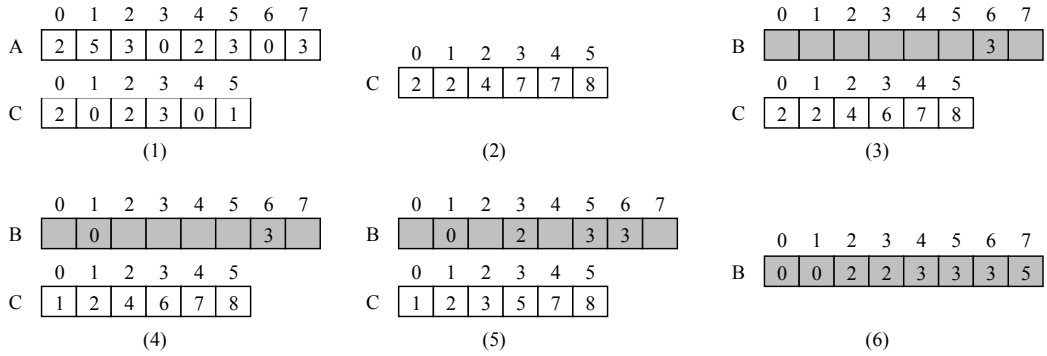


图 9-12 计数排序

9.7.2 基数排序

基数排序是一种“低位优先”的排序方法，它的基本思想是通过反复的对子关键字排序来完成排序。假设元素r[i]的关键字为key_i，key_i是由d位十进制数组成，即$key_i=ki^1\ ki^2 \ldots ki^d$，则每一位可以视为一个子关键字，其中$ki^1$是最高位，$ki^d$是最低位，每一位的值都在 0 到 9 的范围内，此时基数rd =10。如果ki^1是由d个英文字母组成，即$key_i=ki^1\ ki^2 \ldots ki^d$，其中'a' ≤ ki^j ≤'z'（1≤ j ≤d），则基数rd = 26。

排序时先按最低位的值对元素进行初步排序，在此基础上再按次低位的值进行进一步排序。依次类推，由低位到高位，每一趟都是在前一趟的基础上，根据关键字的某一位对所有元素进行排序，直到最高位，这样就完成了计数排序的全过程。

下面我们先通过一个例子来说明基数排序的基本过程。假设对 7 个元素进行排序，每个元素的关键字是 1000 以下的正整数。在此，每个关键字由三位子关键字构成$k^1\ k^2\ k^3$，k^1代表关键字的百位，k^2代表关键字的十位，k^3代表关键字的个位，基数rd = 10。排序的过程如图 9-13 所示。

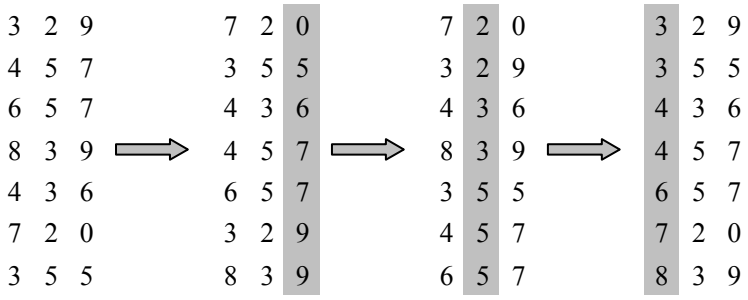


图 9-13 基数排序

需要注意的是，在基数排序的过程中，当针对每一个子关键字进行排序时，需要使用稳定的排序方法。当基数 rd 不太大时，对于每个子关键字的排序而言，计数排序是一种很好的选择，因为其他稳定的排序方法时间复杂度太高。

从算法的执行过程中可以看出，对 n 个具有 d 位子关键字的元素进行排序，每一位子关键字的排序采用计数排序时，需要Θ(n + rd)的时间；排序一共进行 d 趟，因此基数排序的时间复杂度 T(n) = Θ(d(n + rd))。当 d 为常数，且 rd 不太大，即 rd = O(n)时，基数排序可以在现行时间内完成。