第 8 章　查找算法

教学要点

查找操作是在特定的数据集合中寻找满足某种给定条件的数据元素的过程，简单地说，是按数据的内容找到数据对象，这是数据处理中经常要进行的一种操作。因此查找是程序设计中的一项重要的基本技术。

本章介绍查找操作相关的基本概念，讨论若干适用于不同数据结构的经典查找技术，如线性表的顺序查找、二分查找和分块查找算法，二叉排序树的查找算法以及哈希表的查找算法；此外还将分析、比较各种查找算法所适用的存储结构和效率。

本章在 Visual Studio 中用名为 search 的类库型项目实现有关数据结构与算法的基础类定义，用名为 searchtest 的应用程序型项目实现测试和演示程序。

建议本章授课 6 学时，实验 4 学时。

8.1　查找与查找表

在生活、学习和工作中，人们经常要进行某种查找操作，例如，在字典中查找单词，在电话簿中查找电话号码等。与此类似，在数据处理中，常常需要在一组数据中寻找满足某种给定条件的数据元素，这种查找操作是经常使用的一种重要运算。

8.1.1　查找操作相关基本概念

1. 关键字、查找操作、查找表与查找结果

关键字（key）是数据元素类型中用于识别不同元素的某个域（字段），能够唯一地标识数据元素的关键字称为主关键字（primary key）。查找（search）操作是在特定的数据结构中寻找满足某种给定条件的数据元素的过程，这里所谓的满足某种给定条件的数据元素，一般是指它的关键字等于特定的值。

被实施查找操作的数据结构称作查找表（search table），它一般是同一种数据类型的数据元素的有限集合。查找表可以是各种不同的数据结构，如表 8-1 所示的通信簿可以看成是一个顺序存储结构的线性表，这样的通信簿称为顺序查找表。树结构和图结构也常是实施查找操作的对象。例如，主流的计算机文件系统是一个树型结构的数据集合，目录、子目录是树中的分支节点，文件是树的叶子结点，可以在文件系统中以文件名、文件长度、日期等作为关键字查找特定的文件，此时文件系统成为树形查找表。

查找操作是在查找表中，根据给定的某个值 k，确定关键字与 k 相同的数据元素的过程，所以，查找操作也可以说是按关键字的内容找到数据元素。通过查找操作可以查询某个特定数据元素是否在查找表中，或检索查找表中某个特定数据元素的属性。查找的结果有两种：查找成功与查找不成功。如果在查找表中，存在关键字与 k 相同的数据元素，则查找成功；否则，查找不成功。

例如，在表 8-1 所示的通信簿中，以姓名为关键字，查找是否有“刘胜利”的记录，即待查关

键字 k =“刘胜利”。最简单的查找过程是：从查找表的第 1 个数据元素开始依次比较当前记录的关键字和 k 的值是否相等，因第 3 个数据元素的姓名与 k 相同，则查找成功。如果设待查关键字 k =“李伟”，通信簿中所有数据元素的姓名都不等于 k，则查找不成功。

表 8-1 通信簿

姓名	电话号码	电子邮箱
王红	785386	wh@126.cn
张小虎	684721	zxh@whu.edu.cn
刘胜利	1367899	lsl@pku.edu.cn
李明	678956	lm@whu.edu.cn

2. 静态查找表与动态查找表

对查找表除了进行查询和检索操作外，也可能进行其他的操作，如在查找表中插入新的数据元素或删除已存在的数据元素。根据查找表数据是否变化，可以将查找表分为静态查找表和动态查找表：

- 静态查找表（static search table）：不需要对一个查找表进行插入、删除操作，仅作查询和检索操作，例如，字典是我们经常使用的一种工具，我们在字典中查找时，不需要进行诸如插入、删除等操作，所以字典可以视为是一个静态查找表。
- 动态查找表（dynamic search table）：需要对一个查找表进行插入、删除操作，例如，一本个人电话簿在使用的过程中，经常需要增加或删除数据元素，所以，电话簿可以视为是一种动态查找表。

3. 查找方法

一般情况下，数据元素在查找表中所处的存储位置与它的内容无关，那么按照内容查找某个数据元素时不得不进行一系列值的比较操作。

查找的方法一般因数据的逻辑结构及存储结构的不同而变化。一般而言，如果数据元素之间不存在明显的组织规律，则不便于查找。为了提高查找的效率，需要在查找表的元素之间人为地附加某种确定的关系，亦即改变查找表的结构，如先将数据元素按关键字值的大小排序，就可以实施高效的二分查找算法。

查找表的规模也会影响查找方法的选择。对于数据量较小的线性表，可以采用顺序查找算法。例如，从个人电话簿的第一个数据元素开始，依次将数据元素的关键字与待查关键字 k 比较，进行查找操作。

当数据量较大时，顺序查找算法执行效率很低，这时可采用分块查找算法。例如，在词典中查找单词，从头开始进行顺序查找的方法效率低、速度慢，恐怕没有人会经常以这种方式在词典中查找特定的词汇。一般我们会分两步来查找某个特定的单词：首先确定单词首字母的起始页码，再依次根据单词后几个字母的内容，就可以快速准确地定位单词，并查阅其含义。这是因为词典是按字母顺序分块排列的，分块查找可以大大地缩小查找范围。

顺序查找是在数据集合中查找满足特定条件的数据元素的基本方法，要提高查找效率，可先将数据按一定方式整理存储，如排序、分块索引等。所以完整的查找技术包含存储（又称造表）和查找两个方面。总之，要根据不同的条件选择合适的查找方法，以求快速高效地得到查找结果。本章将讨论若干种经典的查找技术。

4. 查找算法的性能评价

查找的效率直接依赖于数据结构和查找算法。查找过程中的基本运算是关键字的比较操作，衡量查找算法性能的最主要标准是平均查找长度（Average Search Length, ASL）。平均查找长度是指查找过程所需进行的关键字比较次数的期望值，即有

$$ASL = \sum_{i=0}^{n-1} p_i \times c_i$$

其中，p_i 是要查找的数据元素出现在位置 i 处的概率，被查找的数据元素处在查找表中不同的位置 i，则查找相应数据元素需进行的关键字比较次数往往是不同的，用 c_i 表示关键字比较次数，n 是查找表中数据元素的个数。

一般要查找的数据元素的出现概率分布 p_i 很难精确确定，通常考虑等概率出现的情况，即对于 n 个可能出现的位置，可设 p_i=1/n。还需区别查找成功和查找不成功的平均查找长度 ASL，因为它们通常不同，分别用 $ASL_{成功}$和 $ASL_{不成功}$表示。

8.1.2　C#内建数据结构中的查找操作

C#语言亦即.NET Framework 的类库中定义了许多使用方便的数据结构，我们以 Array、ArrayList 和 List 以及 Hashtable 和 Dictionary 为例介绍其中的查找操作方面的内容。

1. Array、ArrayList 和 List

C#中的数组都是继承自 System.Array 基类型。Array 类提供了许多用于排序、查找和复制数组的方法。System.Collections.ArrayList 类和 System.Collections.Generic.List（泛型）类都提供了一种使用大小可按需动态增加的数组。

给定某一数组元素的下标（Index）找到该元素的值也可以看成是一种查找操作，它的时间复杂度为 $O(1)$。而找到具有特定值的某元素的过程才是一般意义下的查找操作，最基本的查找方法是：对于给定待查关键字 k，从数组的一端开始，依次与每个数据元素的关键字进行比较，直到查找成功或不成功。

Array，List 和 ArrayList 类都提供了多种重载(overloaded)形式的 IndexOf( )方法实现查找操作，Array 类的 IndexOf 方法是静态方法，它们分别具有下列形式：

1）public static int IndexOf<T>(T[ ] ar, T k);

T 为数组元素的泛型类型，在调用时指明具体的类型。参数 ar 为要搜索的数组；k 为要查找的对象。如果在整个 ar 数组中找到 k 的匹配项，则返回值为第一个匹配项的索引；否则返回值为-1。

2）public static int IndexOf<T> (T[ ] ar, T k, int startindex, int count);

返回给定数据在数组指定范围内首次出现的位置。如果在 ar 数组中从 startIndex 开始并且包含 count 个的元素的这部分元素中找到 k 的匹配项，则返回值为第一个匹配项的索引；否则返回值为-1。

对于已按关键字的值排序好的数据结构，Array 类中提供了实现更高效的二分查找（binary search）技术的 BinarySearch 方法：

1）public static int BinarySearch<T> (T[ ] ar, T k);

T 为数组元素的泛型类型，在调用时指明具体的类型。参数 ar 为要搜索的已排序一维数组；k 为要搜索的对象。返回给定数据在数组中首次出现位置。如果找到 k，则返回值为指定数组 ar 中的值等于 k 的元素的索引。如果找不到 k，则返回值为一个负数 r，它的反码(又称按位补码)i（即 i = ~r）正好是将 k 插入 ar 数组并保持其排序的正确位置。即，如果 k 小于数组 ar 中的一个元素，则返回数组中大于 k 的第一个元素的索引 i 的按位补码 r。r 和 i 之间存在如下关系：i = ~r = -r -1，r = ~i = -i -1。如果 k 大于数组 ar 中的所有元素，则返回最后一个元素的索引加 1 的按位补码。

根据返回值的规则，如果返回值 r < 0，则说明数组 ar 中没有查找的数据 k；如果返回值 r = -1，则说明 k < ar[~r] = ar [0]；如果~r = ar.Length，则说明 k > ar[~r - 1] = ar[ar.Length -1]；其他情况则有 ar[~r - 1] < k < ar[~r]。

2）public static int BinarySearch <T>(T[ ] ar, int startindex, int count, T k);

返回给定数据在数组指定范围内首次出现位置。如果找到 k，则返回值为指定 ar 中的指定 k 的索引。如果找不到 k，返回值的含义同上所述。

2. Hashtable 和 Dictionary

.NET Framework 的类库中定义了名为 Hashtable 的类来表示“<键，值>对”的集合，它定义在 System . Collections 名字空间中。Hashtable 类提供了表示<键，值>对（Key-Value Pair）的集合，集合中的每个元素都是一个存储在 DictionaryEntry 对象中的键/值对，这些（键，值）对根据键的哈希码进行组织。Hashtable 集合内的元素可以直接通过键来索引，如下例中用 tpbook[“王红”]可以得到它的值“785386”。

在.NET Framework 2.0 版本的类库中新增了类似于 Hashtable 类的 Dictionary 泛型类(在 System . Collections . Generic 名字空间中)，它也表示（键，值）对的集合。Dictionary 泛型类是作为一个哈希表来实现的，它提供了从一组键到一组值的映射，通过键来检索值的速度是非常快的，时间效率接近于 O(1)。

通过下面的例子来看看 Dictionary 类的应用方法。

【例8.1】 创建并初始化 Dictionary 以及打印出其值。

```
using System;
using System.Collections.Generic;
    class SamplesDictionary {
        public static void Main() {
            Dictionary<string, string> tpbook =  new Dictionary<string, string>();
            tpbook.Add("王红", "785386");
            tpbook.Add("张小虎", "684721");
            tpbook.Add("刘胜利", "1367899");
            tpbook.Add("李明", "678956");
            tpbook["王浩"] = "678912";
            foreach (KeyValuePair<string, string> kvp in tpbook) {
                Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
            }
            Console.WriteLine("\nRemove(\"王浩\")"); tpbook.Remove("王浩");
            if (!tpbook.ContainsKey("王浩")) {
                Console.WriteLine("Key \"王浩\" is not found.");
            }
        }
    }
```

程序运行结果如下：

```
Key = 王红, Value = 785386
Key = 张小虎, Value = 684721
Key = 刘胜利, Value = 1367899
Key = 李明, Value = 678956
```

```
Key = 王浩, Value = 678912
Remove("王浩")
Key "王浩" is not found.
```

8.2　线性表查找技术

顺序查找是在数据集合中查找满足特定条件的数据元素的基本方法，针对线性表的查找操作有三种基本方法：顺序查找，二分查找和分块查找。要根据不同的条件选择合适的查找方法，以求快速高效地得到查找结果。

8.2.1　顺序查找

在线性表中进行查找的最基本算法是顺序查找（sequential search），又称为线性查找（linear search）。为了查找关键字值等于 *k* 的数据元素，从线性表的指定位置开始，依次将 *k* 与每个数据元素的关键字进行比较，直到查找成功，或到达线性表的指定边界时仍没有找到关键字等于 *k* 的数据元素，则查找不成功。在第 2 章介绍线性表时，我们在顺序表和链表中都实现了顺序查找算法，它们都以方法 IndexOf 的某种重载形式提供。本章将第 2 章中定义的顺序表 SequencedList 类进行修改，以定义顺序查找表 LinearSearchList 类，重点突出其中的查找操作。

1. 顺序查找的基本思想

设已在顺序查找表 LinearSearchList 对象中保存了一组数据元素，在其中查找一个给定值 *k*，可以采用如下所述的顺序查找算法：

1) 初始化，令 *i*=0。
2) 进入循环：比较序号为 *i* 的数据元素的关键字是否等于 *k*，如果相等，则查找成功，查找过程结束；否则 *i* 自增 1，即 *i*++，继续比较直至查找表中的所有元素。
3) 如果查找成功，则算法返回关键字值等于 *k* 的元素序号 *i*；如果线性表中所有数据元素的关键字都不等于 *k*，则查找不成功，算法返回-1。

顺序表的顺序查找过程如图 8.1 所示。

图 8.1　顺序存储线性表的顺序查找过程

2. 顺序查找表的定义

顺序查找表定义为 LinearSearchList 类，它实现顺序表的顺序查找操作及其他基本操作。查找表元素的类型仍设计为泛型，但要求是可进行比较操作的类型（由子句 where T: IComparable 指示）。

```
public class LinearSearchList<T> where T: IComparable {
    private T[] items;                          // 存储数据元素的数组
    private int count;                          // 顺序表的长度
    int capacity = 0;                           // 顺序表的容量

    public LinearSearchList(int capacity) {
        items = new T[capacity];                // 分配capacity个存储单元
        count = 0;                               // 此时顺序表长度为0
        this.capacity = capacity;
    }
    public LinearSearchList() : this(16) { }

    public void Add(T k) {                      // 将k 添加到顺序表的结尾处
        if (Full) {                             // resize array
            Capacity = capacity * 2;            // double capacity
        }
        items[count] = k; count++;
    }

    public bool Full { get { return count >= capacity; } } // 判断顺序表是否已满

    public int Count { get { return count; } }     // 返回顺序表长度

    public int Capacity {
        get { return capacity; }
        set {
            if (value > capacity) {
                capacity = value;
                T[] copy = new T[capacity];      // create newly sized array
                Array.Copy(items, copy, count); // copy over the array
                items = copy;  // assign items to the new, larger array
            }
        }
    }

    public void Show(bool showTypeName) {
        if (showTypeName)
            Console.Write("LinearSearchList: ");
        for (int i = 0; i < this.count; i++) {
            Console.Write(items[i] + "  ");
```

```
            }
            Console.WriteLine();
        }
    }
```

顺序查找算法实现在方法 IndexOf 和 Contains 中，编码如下：

```
        // 查找k值在线性表中的位置,查找成功时返回k值首次出现位置，否则返回-1
        public int IndexOf(T k) {
            int i = 0;
            while (i < count && !items[i].Equals(k) )
                i++;
            if (i >= 0 && i < count)
                return i;
            else return -1;
        }
        //  查找线性表是否包含k值,查找成功时返回true，否则返回false
        public bool Contains(T k) {
            int j = IndexOf(k);
            if (j != -1)
                return true;
            else
                return false;
        }
```

3. 单向链表中的查找操作

如果数据保存在单向链表中，对于一个给定值 k，顺序查找就是从链表的第一个数据结点开始，沿着链接的方向依次与各结点数据进行比较，直至查找成功，或表中所有数据元素的关键字都不等于 k，则查找不成功。在第二章介绍的 SingleLinkedList 类中增加实现顺序查找算法的 IndexOf 方法如下：

```
        public int IndexOf(T k) {
            int i = 0;
            SingleLinkedNode<T> q = head.Next;
            while (q != null) {
                if (k.Equals(q.Item))
                    return i;
                q = q.Next;
                i++;
            }
            return -1;
        }
```

4. 算法分析

同前面的章节保持一致，在下面的分析中假定线性查找表中的元素序号从零开始。

设线性查找表的长度为 n，查找第 i 个元素的概率为 p_i，假设为等概率条件，即 $p_i = 1/n$。如果线性查找表中位置 i 处的元素的关键字等于 k，进行 i+1 次比较即可找到该元素。

对于成功的查找，关键字的平均查找长度 $\mathrm{ASL}_{成功}$ 为

$$\mathrm{ASL}_{成功}=\sum_{i=0}^{n-1}p_i\times c_i=\frac{1}{n}\sum_{i=0}^{n-1}(i+1)=\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$$

每个不成功的查找，都只有在 n 次比较后才能确定，故关键字的平均查找长度 $\mathrm{ASL}_{不成功}$ 为 n，即

$$\mathrm{ASL}_{不成功}=\sum_{i=0}^{n-1}p_i\times c_i=\sum_{i=0}^{n-1}\frac{1}{n}\times n=n$$

可见，在等概率条件下，查找成功的平均查找长度约为线性表长度的一半，查找不成功的平均查找长度等于线性表中元素的个数。由此可知，顺序查找算法的时间复杂度为 $O(n)$。

8.2.2 二分查找

如果顺序存储结构的数据元素已经按照关键字值的大小排序，则可以在其上进行二分查找（binary search），二分查找又称折半查找。

1. 二分查找的基本思想

不失一般性，假定线性表的数据元素是按照升序排列的，对于待查关键字值 k，从线性表的中间位置开始比较，如果当前数据元素的关键字等于 k，则查找成功，返回查找到的数据元素的序号。否则，若 k 小于当前数据元素的关键字，则以后在线性表的前半部分继续查找；反之，则在线性表的后半部分继续查找。依照同样的原理重复进行这一过程，直至全部数据集搜索完毕，如果仍没有找到，则说明查找不成功，返回一个负数。

以如下数据序列（仅考虑数据元素的关键字）为例：

{1, 3, 14, 19, 23, 32, 55, 86}

假设要查找 k=23，二分查找算法描述如下：

1) 初始化。令变量 left 和 right 分别为查找范围的左边界和右边界，即 left=0, right=7, 计算变量 mid = (left + right) / 2，即设置 mid 是 left 和 right 的平均数（取整），此时 mid=3，将线性表下标为 mid 的数据元素与 k 进行比较，即比较 items[mid]与 k，如图 8.2(a)所示。
2) 因为 k> items[mid]，故以后只需在线性表的后半部继续比较，查找范围缩小，left 上移，right 不变，即令 left=mid+1=4，right=7，并更新 mid=5，如图 8.2(b)所示。
3) 再比较 k 与 items[mid]，有 k< items[mid]，说明以后只需在 mid 的前半部继续比较，查找范围缩小。left 不变，right 下移，即令 right=mid-1=4，并更新 mid=4，如图 8.2(c)所示。此时若 k == items[mid]，则查找成功，mid 为查找到的数据元素的序号，将其值返回。
4) 如果经过多次移动 left 和 right，使得 left>right，说明查找不成功，返回一个负数 r 来标明查找不成功，这个负数的反码(又称按位补码)i 正好是将 k 插入线性表并保持其排序的正确位置。

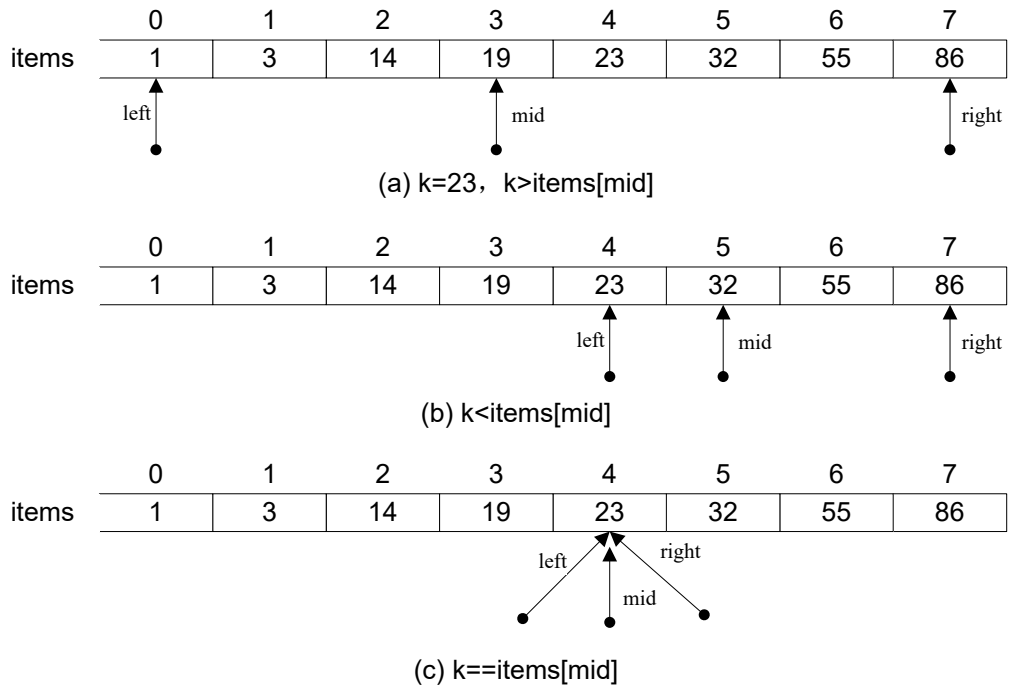

图 8.2　二分查找的过程

2. 二分查找算法实现

在顺序查找表 LinearSearchList 类中增加实现二分查找算法的 BinarySearch 方法。该方法的参数和返回值与前面介绍的 Array 类中的 BinarySearch 方法相同。

```
// 查找k值在线性表中的位置，查找成功时返回k值首次出现位置，否则返回应插入位置的反码
// 查找范围:从下标si开始，包含length个元素
public int BinarySearch(T k, int si, int length) {
    int mid = 0, left = si;
    int right = left + length - 1;
    while (left <= right) {
        mid = (left + right) / 2;
        if (k.CompareTo(items[mid]) == 0)
            return mid;
        else if (k.CompareTo(items[mid]) < 0)
            right = mid - 1;
        else
            left = mid + 1;
    }
    if (k.CompareTo(items[mid]) > 0)
        mid++;
    return ~mid;
}
```

【例8.2】　创建顺序查找表，对其进行排序后测试二分查找算法。

```
using System; using DSAGL;
namespace searchtest {
```

```
class LinearSearchListTest {
    static void Main(string[] args) {
        int n = 10;
        LinearSearchList<int> sl = new LinearSearchList<int>(n+8);
        Randomize(sl, n);
        Console.Write("随机排列： "); sl.Show(true);
        Console.Write("排序后： "); sl.Sort(); sl.Show(true);
        int k = 50;
        int re = sl.BinarySearch(k, 0, n);
        Console.WriteLine("k={0}, re={1}, i={2}", k, re, ~re);
    }
    // 用0到100之间的随机整数填充线性表
    static void Randomize(LinearSearchList<int> sl, int n) {
        int k; Random random = new Random();
        for (int i = 0; i < n; i++) {
            k = random.Next(100);
            sl.Add(k);
        }
    }
}
}
```

程序运行结果如下：

```
随机排列： LinearSearchList: 73  56  79  53  79  70  99  99  29  36
排序后 ： LinearSearchList: 29  36  53  56  70  73  79  79  99  99
k=50, re=-3, i=2
```

结果说明，在随机产生的一组数中不包含 50 这个数；如果要将 50 插入这个排好序的组数中，应该将它插入到 2 号位置，即 36 和 53 之间。

3. 算法分析

在长度 n=8 的线性表中进行二分查找的过程如图 8.3 所示，二分查找过程形成一棵二叉判定树。结点中的数字表示线性表中数据元素的下标。

二叉判定树反映了二分查找过程中进行关键字比较的数据元素次序和操作的推进过程。当 n=8 时，线性表的左边界为 0，即 left=0，右边界为 7，即 right=7，第一次 k 与下标为 mid=(0+7)/2=3 的数据元素比较，若 k 值较小，再与下标为 1 的数据元素比较，否则与下标为 5 的数据元素比较；继续查找依照同样的方法。

设二叉判定树的高度为 h，则 h 满足下式：

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

在二叉判定树中，一次成功的查找将走过一条从根结点出发到二叉树中的某结点结束的路径，进行比较操作的次数为这条路径所经过的结点个数，最少为 1 次，最多为$\lceil \log_2(n+1) \rceil$，平均比较次数与查找表元素个数 n 的关系为$O(\log_2 n)$。

不成功的查找路径则总是从根结点到某个叶子结点，平均比较次数与查找表元素个数 n 的关系也为$O(\log_2 n)$。

因此，二分查找算法的时间复杂度为$O(\log_2 n)$。二分查找算法每比较一次，如果查找成功，算法结束；否则，将查找的范围缩小一半。而顺序查找算法在每一次比较后，仅将查找范围缩小了一个数据元素。可见，二分查找算法的平均效率比顺序查找算法高。

顺序查找算法简单，对原始数据不要求已排序，适用于顺序存储结构和链式存储结构；二分查找算法虽然减少了查找次数，速度较快，但条件严格，要求数据序列是顺序存储并且已排序的，而对数据序列进行排序也是要花费一定的时间代价的。

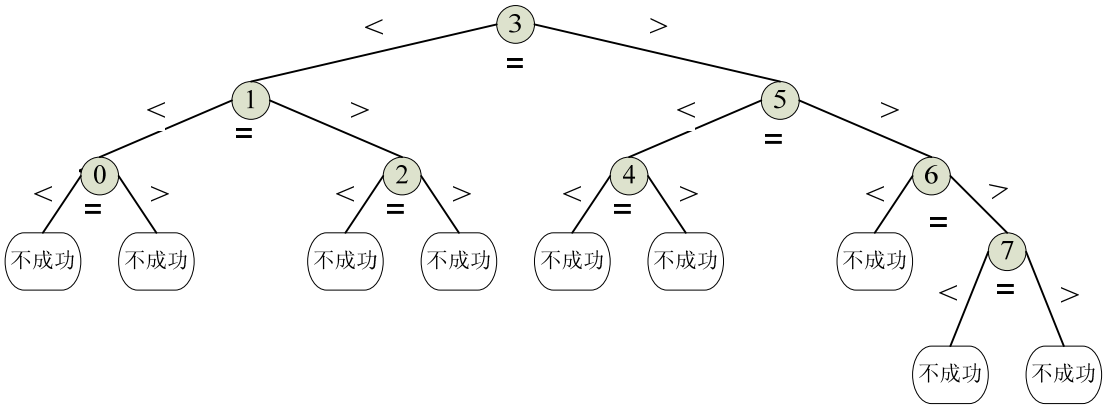

图 8.3 二分查找过程的二叉判定树

8.2.3 分块查找

当数据量较小时，可以采用前面介绍的顺序查找算法；但当数据量较大时，顺序查找的效率比较低,操作所需花费的时间可能比较多。在一定条件下可以采用分块查找（blocking search）算法来提高查找速度。例如，在字典中查找单词一般都使用了分块查找方法，这是因为字典中的单词是按字母顺序分块排列的。

1. 分块查找的基本思想

要对数据进行分块查找，首先需要将数据分块存储，即将数据序列中的数据元素存储在若干数据块中，数据块按照数据元素的关键字大小排序，而在每一个数据块内，各数据元素可以是排序的，也可以未排序，这种分块特性称为“块间有序”。

另设一个索引表（index table），记录每个数据块的起始位置。通过索引表的帮助，迅速缩小查找的范围。

不失一般性，可以假定不同的数据块按照数据元素关键字的递增次序排列，亦即，处于较前面的块中的任意一个数据元素的关键字都小于后面块中的所有数据元素的关键字。

例如，字典可以看成是数据量较大的查找表，使用顺序查找方法来查字典显然是不合适的，适宜的方法是采用分块查找技术。为使查找方便，字典中的单词通常是按照字母顺序分块排列的，并且字典都有一个索引表指明每个数据块的起始位置，通过索引表的帮助，对一个单词的查找，就能限定到一个特定的块中较快地完成。

2. 静态查找表的分块查找

字典是一种典型的静态查找表，主要操作是查找，不需要进行诸如插入、删除等操作，可以采用顺序存储结构来存储数据。

字典分块查找算法的基本思想是：将所有单词排序后存放在数组 dict 中，并为字典设计一个索引

表（index），记录每个数据块的起始位置，即 index 中的每个元素由两部分组成：首字母和块起始位置下标，它们分别对应于单词的首字母和以该字母为首字母的所有单词在 dict 数组中的起始下标。

通过索引表 index，将较长的单词表 dict 在逻辑上划分成若干个数据块，以首字母相同的若干单词构成各自的数据块，每块的起始位置记录在索引表 index 中，由 index 中对应于“首字母”列的“块起始位置下标”列标明。

在字典 dict 中查找给定的单词 token，使用分块查找算法包括下面两个步骤：

1) 在索引表 index 中查找 token 的首字母，以确定 token 在 dict 中的哪一个数据块。

2) 跳到相应数据块中，使用顺序或二分查找方法在块内查找 token。

在某一数据块内进行的顺序查找，可以通过顺序查找表中的重载方法 IndexOf 来完成，通过提供更多的参数以限定查找范围。该方法具有下列形式：

```
public int IndexOf(T k, int si, int length) {
    int j = si;
    while( (j < si+length) && !items[j].Equals(k) )
        j++;
    if(j >= si && j < si + length)
        return j;
    else return -1;
}
```

参数 si 和 length 用来限定查找范围，一般通过在索引表 index 中查找所在块的信息来确定调用 IndexOf 方法时给这些参数所赋的值。

3. 动态查找表的分块查找

动态查找表中的主要操作除查找外，还经常需要对查找表进行插入、删除操作。动态查找表的存储结构必须适应插入或删除操作给数据集带来的动态变化。

例如，个人电话簿是一种动态查找表。如果以顺序存储结构保存电话簿的数据，则进行插入、删除操作时必须移动大量的数据元素，运行效率低。如果以链式存储结构保存电话簿的数据元素，虽然插入、删除操作比较方便，但相应的缺点是，不仅花费的空间比较多，而且查找的效率也比较低。

以顺序存储结构和链式存储结构相结合的方式来存储数据元素，就可能既最大限度地利用空间，又有很高的运行效率。

【例8.3】 创建动态查找表，对其测试分块查找算法。

不失一般性，设每个数据元素仅由关键字组成。对于如下的数据序列：

{10, 6, 23, 5, 2, 26, 33, 36, 43, 41, 40, 46, 49, 57, 54, 53, 67, 61, 71, 74, 72, 89, 80, 93, 92}

采用如图 8.4 所示的分块存储结构。定义 DynamicLinearBlockSearchList 类表示动态分块查找表。

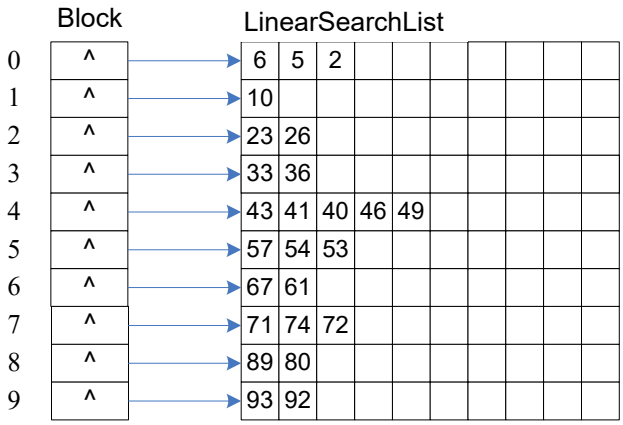

图 8.4　动态查找表的分块存储结构

类成员 Block 数组充当各数据块的索引表，元素 Block[i]指向一个数据块，它的类型为前面定义的顺序查找表 LinearSearchList 类。设每个数据块最多可保存 10 个数据元素，Block[0]指向的数据块保存值为 0 到 9 的数据元素，Block[1]指向的数据块保存值为 10 到 19 的数据元素，依次类推。

Block 数组的元素保存对数据块的引用，它们的初值均为 null。在向查找表增加数据元素时，根据实际需要，在已有数据块中添加数据或动态生成新的数据块，将对数据块的引用保存在 Block 数组相应的元素中。

```
public class DynamicLinearBlockSearchList{
    private LinearSearchList<int>[] Block;
    private int Blocksize;

    public DynamicLinearBlockSearchList(int capacity, int bs) {
        Blocksize = bs;
        int nb = (capacity % bs == 0)? capacity/bs: capacity/bs+1;
        Block = new LinearSearchList<int>[nb];
    }

    public void Show() {
        for (int i = 0; i < Block.Length; i++) {
            Console.Write("Block [" + i + "]");
            if (Block[i] == null)
                Console.WriteLine(" = .");
            else {
                Console.Write("->");
                Block[i].Show(true);
            }
        }
        Console.WriteLine();
    }
}
```

在动态查找表 DynamicLinearBlockSearchList 类中，Contains 方法在表中查找给定值，Add 方法在表中插入结点。

Add(int k)将数据元素*k*添加到查找表中。它首先根据*k*的值确定应该添加到由Block[k/Blocksize]指向的数据块，再调用 LinearSearchList 类的 Add(int k)方法将数据元素 *k* 添加到相应的数据块中。

```
public void Add(int k) {
    int i = k / Blocksize;
    if (Block[i] == null) {
        Block[i] = new LinearSearchList<int>(Blocksize);
    }
    Block[i].Add(k);
}
```

Contains(int k)方法实现分块查找算法。首先根据分块规则，确定可能所属的数据块，再调用 LinearSearchList 类的 Contains (int k)方法在相应的数据块中求得查找结果。

```
public bool Contains(int k) {
    int i = k / Blocksize;
    bool found = false;
    if (Block[i] != null) {
        Console.Write("search k=" + k + " in Block[" + i + "]\t");
        found = Block[i].Contains(k);
    }
    return found;
}
```

DynamicLinearBlockSearchList 类的测试程序如下所示：

```
class DynamicBlockSearchTest {
    static void Main(string[] args) {
        DynamicLinearBlockSearchList sl = new DynamicLinearBlockSearchList(100, 10);
        Randomize(sl, 25);          //在查找表中添加25个随机数
        sl.Show();
        bool f = sl.Contains(50);
        Console.WriteLine("Contains(" + 50 + ")=" + f);
    }

    static void Randomize(DynamicLinearBlockSearchList sl, int len) {
        int k;
        Random random = new Random();
        for (int i = 0; i < len; i++) {
            k = random.Next(100);
            sl.Add(k);
        }
    }
}
```

程序运行结果如下：

```
Block[0]->LinearSearchList: 6  5  2
Block[1]->LinearSearchList: 10
```

```
Block[2]->LinearSearchList: 23  26
Block[3]->LinearSearchList: 33  36
Block[4]->LinearSearchList: 43  41  40  46  49
Block[5]->LinearSearchList: 57  54  53
Block[6]->LinearSearchList: 67  61
Block[7]->LinearSearchList: 71  74  72
Block[8]->LinearSearchList: 89  80
Block[9]->LinearSearchList: 93  92
search k=50 in Block[5]    Contains(50)=False
```

8.3　二叉查找树及其查找算法

在数据集合中查找满足特定条件的数据元素，顺序查找是基本方法。要提高查找效率，可先将数据按一定方式整理存储，如以某种有序的方式存储在树结构中，可能会大大提高后续的查找操作的效率或方便实施其他操作。本节以二叉查找树（Binary Search Tree, BST）为例，介绍二叉树结构的查找算法。在普通的二叉树中查找，可能需要遍历整棵二叉树，而在二叉查找树中查找，进行查找所产生的比较序列仅是搜索二叉树中的一条路径，不需要遍历整棵二叉树。

1. 二叉查找树的定义

二叉查找树具有下述性质：

- 如果根结点的左子树非空，则左子树上所有结点的关键字值均小于等于根结点的关键字值。
- 如果根结点的右子树非空，则右子树上所有结点的关键字值均大于根结点的关键字值。
- 根结点的左、右子树也分别为二叉查找树。

二叉查找树又称二叉排序树。根据上述性质可知，二叉排序树的中根遍历序列是按升序排列的。例如，以数值序列

{6, 3, 2, 5, 8, 1, 7, 4, 9}

建立的一棵二叉查找树如图 8.5 所示，它的中根遍历序列是{1, 2, 3, 4, 5, 6, 7, 8, 9}。

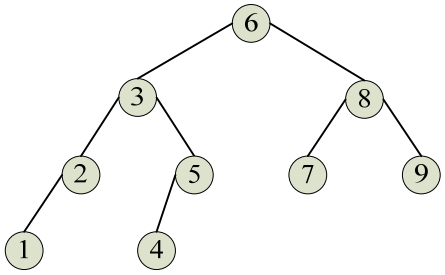

图 8.5　一棵二叉查找树

定义二叉查找树类 BinarySearchTree，它继承第六章中定义的二叉树类 BinaryTree，结点为 BinaryTreeNode 类。

```
public class BinarySearchTree<T> : BinaryTree<T> where T:IComparable{ ..... }
```

二叉查找树类是数据集合类型，其元素类型设计为泛型，是为了适应构造各种数据类型的集合实例，但要求元素属于可比较的类型，上面的子句 where T: IComparable 表达这一要求。

在二叉查找树 BinarySearchTree 类中，Contains 方法在树中查找给定值，Add 方法在树中插入结点，类的构造方法根据给定的数据序列建立二叉查找树。下面依次介绍二叉查找树的查找、插入和建树算法。

2. 在二叉查找树中进行查找

在一棵二叉查找树中查找给定值 *k* 的算法描述如下：

1) 初始化，变量 p 初始指向二叉查找树的根结点。
2) 进入循环，直到查找成功或 p 为空：比较 *k* 和 p 结点的关键字，如果两者相等，则查找成功；若 *k* 值较小，则进入 p 的左子树继续查找；若 *k* 值较大，则进入 p 的右子树继续查找。
3) p 为非空时表示查找成功，p 为空时表示查找不成功。

实现查找算法的代码如下：

```
public bool Contains(T k) {
    BinaryTreeNode<T> p = root;
    Console.Write("search(" + k + ")=  ");
    while (p != null && !k.Equals(p.Data) ) {              //比较是否相等
        Console.Write(p.Data + " ");
        if (k.CompareTo(p.Data) < 0)                       //比较大小
            p = p.Left;                                    //k小，进入左子树
        else
            p = p.Right;                                   //k大，进入右子树
    }
    if (p != null)
        return true;                                       //查找成功
    else
        return false;                                      //查找不成功
}
```

在二叉查找树中，从根结点到某结点所经过的结点序列，正好是查找该结点所进行的一次成功查找。例如，在图 8.5 的二叉查找树中查找 7，比较的结点序列是{6, 8, 7}。而查找不成功的路径，是从根结点到某叶子结点所经过的结点序列。假如要查找 4.5，需要比较的结点序列是{6, 3, 5, 4}，当已与叶子结点 4 比较过并且不相等，才能作出查找不成功的结论。

在一般的二叉树中进行查找，其过程实质是一个遍历二叉树的过程，因为，理论上要将二叉树的每个结点与查找关键字进行比较。但在二叉排序树查找中，为查找而产生的比较操作序列只是搜索二叉树中的一条路径，而不是遍历整棵树，不需要访问所有结点。

3. 在二叉查找树中插入结点

在二叉查找树中插入一个值为 *k* 的结点，使得插入操作的结果仍然是一颗二叉查找树，所以，插入操作首先需要找到新结点应该插入的位置，这是一个查找问题，而且，通常情况下这是一次不成功的查找。插入结点的算法描述如下：

1) 如果是空树，则为数据 *k* 建立一个新结点，并将此结点作为二叉查找树的根结点。
2) 否则从根结点开始，将数据 *k* 与当前结点的关键字进行比较，如果 *k* 值较小，则进入其左子树；如果 *k* 值较大，则进入其右子树。循环迭代直至当前结点为空结点。
3) 为数据 *k* 建立一个新结点，并将新结点与最后访问的结点进行值的比较，插到合适的位置。

按照该算法，每次新插入的结点都是叶子结点，这样就不会破坏二叉查找树原有的形态。

结点的插入算法的实现代码如下：

```
public void Add(T k) {
    BinaryTreeNode<T> p, q = null;
    if (root == null)
        root = new BinaryTreeNode<T>(k);                    //建立根结点
    else {
        p = root;
        while (p != null) {
            q = p;
            if (k.CompareTo(p.Data) <= 0)
                p = p.Left;
            else
                p = p.Right;
        }
        p = new BinaryTreeNode<T>(k);
        if (k.CompareTo(q.Data) <= 0)                       //q为最后访问的结点
            q.Left = p;
        else
            q.Right = p;
    }
}
```

4. 二叉排序树的建立

在二叉查找树 BinarySearchTree 类中用构造方法建立一棵二叉排序树，它将其参数（一个数组）包含的数据依次插入二叉排序树中。

```
public BinarySearchTree(T[] td){              //以数组中的数据建立二叉排序树
   Console.Write("建立二叉排序树：   ");
   for(int i=0;i<td.Length;i++) {
        Console.Write(td[i] + " ");
        Add(td[i]);
   }
   Console.WriteLine();
}
```

以下列关键字序列为例：

{6, 3, 2, 5, 8, 1, 7, 4, 9}

建立一棵二叉查找树的过程如图 8.6 所示。

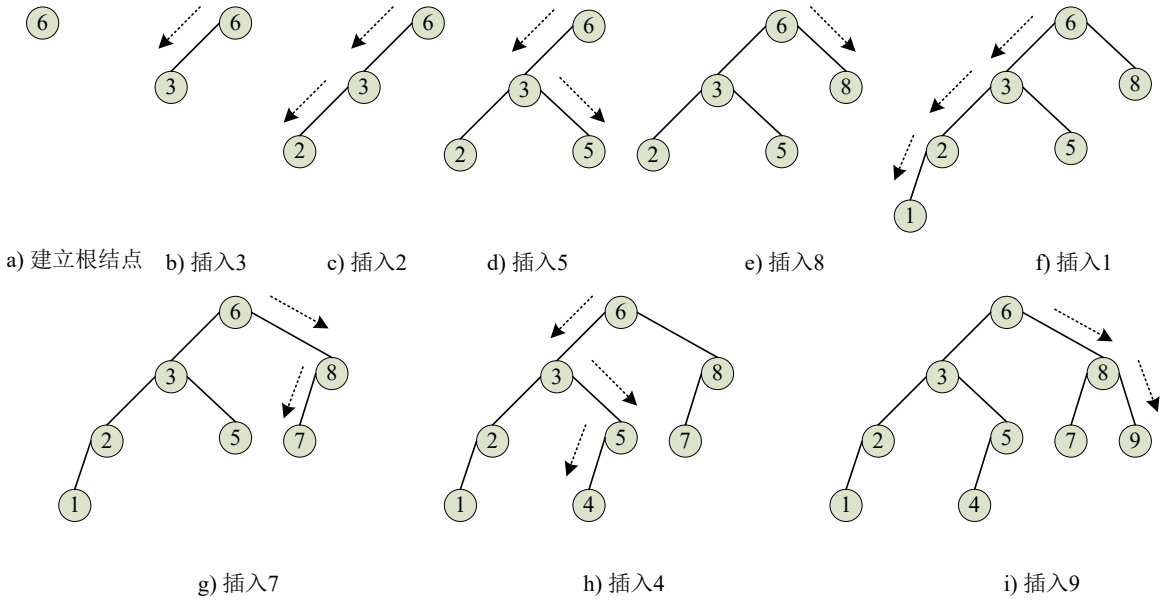

图 8.6 建立二叉查找树

【例8.4】 建立二叉查找树并测试其结果。

BinarySearchTree 类的测试程序如下所示：

```
using DSAGL;
namespace searchtest {
    class BinarySearchTreeTest {
        public static void Main(string[] args) {
            int[] td = { 5, 8, 3, 2, 4, 7, 9, 1, 5 };
            BinarySearchTree<int> tr = new BinarySearchTree<int>(td);
            tr.ShowInOrder();
        }
    }
}
```

程序运行结果如下：

```
建立二叉排序树：  5 8 3 2 4 7 9 1 5
中根次序：  1 2 3 4 5 5 7 8 9
```

8.4 哈希查找

前面介绍的查找算法，无论是顺序查找、二分查找或分块查找算法，都需要进行一系列的关键字值的比较操作才能确定数据元素在查找表中的位置，或得出查找不成功的结论。这些查找算法的平均查找长度 ASL 都与查找表的规模，即表中数据元素的个数有关，数据元素越多，为查找而进行的平均比较次数就越多。在这些查找表中，数据元素所占据的存储位置往往与数据元素的内容本身无关，那么按照内容查找某个数据元素时不得不进行一系列值的比较操作。

如果能做到按数据内容决定存储位置，就有可能高效实施按内容查找数据。与前述诸多查找方法不同，哈希（hash）技术正是一种按关键字的值编址以实现存储和检索数据的方法。哈希意为杂凑，

也称散列，它使用哈希函数（hash function）完成某种关键字集合到地址空间的映射，按哈希方法建立的一组数据存储区域称为哈希表（hash table）。

8.4.1　哈希查找的基本思想

在查找表中，如果数据元素所占据的存储位置与数据元素的关键字值无关，那么查找某个数据元素时不得不进行一系列值的比较。如果能在数据元素的关键字与其存储位置之间建立一种对应关系，就可以通过对关键字的运算直接得到数据元素的存储位置，而不需要进行多次比较，从而提高查找的效率。哈希查找技术就是基于这种思想设计的一种查找方法，哈希技术利用哈希函数确定数据元素的关键字与其存储位置的对应关系。

哈希函数实质上是关键字集合到地址空间的映射，按哈希函数建立的一组数据元素的存储区域称为哈希表。以哈希函数构造哈希表的过程称为哈希造表，以哈希函数在哈希表中查找的过程称为哈希查找。

哈希查找技术的设计思想是，根据数据元素的关键字值 k 计算出相应的哈希函数值 hash(k)，这个函数值决定该数据元素的存储位置。基于这一思想进行哈希造表过程，在造表过程中将待查找的数据序列存储在哈希表中。而在哈希查找过程中，直接计算查找关键字的哈希函数值，以得到数据元素的存储位置或给出查找不成功的信息。

在计算哈希函数时，如果有两个不同的关键字 k_1 和 k_2，对应相同的哈希函数值，表示不同关键字的多个数据元素映射到同一个存储位置。这种现象称为冲突（collision），与 k_1 和 k_2 分别对应的两个数据元素称为同义词。

如果哈希表的存储空间足够大，使得所有数据元素的关键字与其存储位置是一一对应的，则不会产生冲突。但被处理的数据一般来源于较大的集合，而计算机系统的地址空间则是有限的，因此在解决实际问题时，哈希函数一般是从大集合（关键字的定义域）到小集合（地址空间）的映射，冲突是不可避免的。我们一方面要考虑如何尽可能减少冲突，另一方面则要考虑当冲突发生时如何解决冲突。

哈希查找技术包括以下两个关键问题：

- 避免冲突（collision avoidance）：主要是通过设计一个好的哈希函数，尽可能减少冲突。
- 解决冲突（collision resolution）：因为冲突是不可避免的，发生冲突时，需要实施一种有效解决冲突的策略（collision resolution strategy）。

8.4.2　哈希函数的设计

为避免冲突，需设计一个好的哈希函数。哈希函数一般是从大集合（关键字的定义域）到小集合（地址空间）的映射，一个好的哈希函数应该能将关键字值均匀地分布在整个哈希表的地址空间中，这样就尽可能地减少了冲突的机会。哈希函数在值域分布得越均匀，产生冲突的可能性就越小。

为了设计好的哈希函数，应该考虑以下几方面的因素：

- 系统存储空间的大小和哈希表的大小；
- 查找关键字的性质和数据分布情况；
- 数据元素的查找频率；
- 哈希函数的计算时间。

在针对具体问题设计哈希函数时，上述因素需要综合考虑，一般原则是，好的哈希函数应该发挥关键字的所有组成成份的作用，从而充分反映关键字区别不同元素的能力，这样实现的关键字到地址的映射就会比较均匀。

下面介绍设计哈希函数的几种常用方法。

1. 除留余数法

除留余数法较简单，哈希函数设定为

$$\text{hash}(k) = k \% p$$

显而易见，哈希函数的值域为 0～p - 1。在除留余数法定义的哈希函数中，参数 p 有多种取值方法，例如：

1） 选 p 为 10 的某个幂次方；如果选定 $p = 10^3$，哈希函数值即是取关键字值的后三位，亦即数据按其关键字值的后三位编址存储。在这种情况下，后三位相同的所有关键字有相同的哈希函数值，即产生冲突。例如，6123 与 7123 构成同义词，在哈希表中的地址都是 123，因而产生冲突。

2） 选 p 为小于哈希表长度的最大素数。

对于不同的问题，选取不同的 p 值对所产生的哈希表的性能影响是不一样的。

2. 平方取中法

平方取中法将关键字值的平方（k^2）的中间几位作为哈希函数 hash(k)的值，而所取的位数取决于哈希表的长度。例如，k = 381，k^2 = 14 51 61，若表长为 100，取中间两位，则哈希函数 hash(k) = 51。

因为乘积的中间几位数和乘数的每一位都相关，所以平方取中法定义的哈希函数在某些情况下产生冲突的可能性较小。

3. 折叠法

折叠法将组成关键字值的不同成份按照某种规则组合在一起。例如折叠移位法，将关键字分成若干段，高位数字右移后与低位数字相加，得到的结果作为哈希函数值。

不同的查找问题所采用的关键字可能差异很大，每种关键字类型都有自己的特殊性。例如，以整数或字符串作为关键字时，哈希函数的定义方式就应该有所不同。总的来说，不存在一种哈希函数对任何关键字集合都能达到最佳效果的情况。在实际应用中，应该根据具体情况，分析关键字值与地址空间之间可能的映射关系，构造不同的哈希函数，或将若干基本的哈希函数组合起来使用。例如，.NET Framework 类库中 Hashtable 类使用的哈希函数定义为：

hash(key) = {key.GetHashCode()+1+[key.GetHashCode()>>5+1]%(hashsize–1)} % hashsize

其中，GetHashCode 方法继承于 Object 类，可由自定义类型重新定义（override），这样就有可能根据关键字的性质定义合适的哈希函数，以达到最佳效果。

8.4.3 冲突解决方法

一个好的哈希函数能使关键字不同的数据元素在哈希表中的分布较为均匀，但好的哈希函数也只能减少冲突，而不能完全避免冲突。所以，在哈希技术中，当冲突发生时还必须有效解决冲突。

解决冲突的方法有很多，这里介绍探测定址法（probing rehashing）、再散列法（rehashing）和散列链法（hash chaining）。.NET Framework 类库中 Hashtable 类使用再散列法来解决冲突，Dictionary 类使用散列链法解决冲突。

1. 探测定址法

在哈希造表阶段，设关键字为 k 的数据元素的哈希函数值为 i = hash(k)，如果哈希表中位置 i 处为空，则存入该数据元素；否则表明产生了冲突，需在哈希表中探测一个空位置来存入该数据。

探测定址的具体方法有多种，如线性探测、平方探测和随机探测法。下面以最简单的线性探测法

为例对探测定址的基本思想进行说明。在产生冲突时，线性探测法继续探测下一个空位置。当探测了哈希表全部空间而没有找到空位置时，说明哈希表已满，无法再存入新的数据元素，这种情况称为溢出。通常另建一个溢出表来处理溢出的情况，原来的哈希表称为哈希基表。

例如，为如下的关键字序列：

{19, 14, 23, 1, 32, 86, 55, 3, 62, 10}

采用线性试探法进行哈希造表。设哈希函数定义为 hash(k) = k % 7，所生成的哈希基表与溢出表如图 8.7 所示。

在查找过程中，设查找关键字为 k，计算哈希函数值 i = hash(k)，将 k 与哈希基表中位置 i 处的数据元素进行比较，如果相等，则查找成功，否则继续在哈希基表中向后顺序查找。如果在哈希基表中没有找到，还要在溢出表查找，在溢出表中常采用基本的顺序查找。可见，此时哈希查找已蜕变为顺序查找。

线性试探法是一种较原始的方法，简单易行，实现方便，但其中存在的缺陷也很严重，包括以下几点：

- 可能产生溢出现象，必须另行设计溢出表并采取相应的算法来处理溢出现象。
- 容易产生堆积（clustering）现象，即存入哈希表的数据元素连成一片，增大了产生冲突的可能性。
- 哈希表只能查找和插入数据元素，不能删除数据元素。如果删除了某数据元素，将中断哈希造表过程中形成的探测序列，以后将无法查到具有相同哈希函数值的后继数据元素。

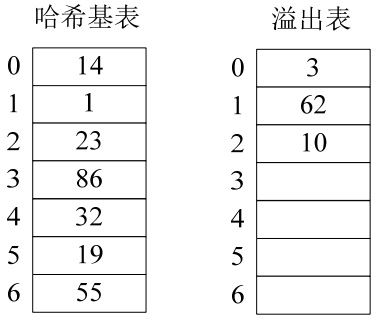

图 8.7 线性试探法的哈希表

2. 再散列法

在再散列法中要定义多个哈希函数：

$$H_i = \mathrm{Hash}_i(key), \quad i = 1, \ldots, n$$

当同义词对一个哈希函数产生冲突时，计算另一个哈希函数，直至冲突不再发生。这种方法不易产生堆积现象，但增加了计算时间。

3. 散列链法

散列链法的基本思想是，所有哈希函数值相同的数据元素，即产生冲突的数据元素，被存储到一个线性链表中，它称为哈希链表；而用一个哈希基表记录所有的哈希链表。散列链法对于冲突的解决既灵活又有效，得到了更多的应用。

散列链法的造表过程是：对于关键字 k，首先计算其哈希函数值 i = hash(k)，如果位置 i 处为空，则存入该数据元素；否则表明产生了冲突，此时创建一个结点存放该数据元素，并将该结点插入到相

应的哈希链表中。

以如下的关键字序列：

{19, 14, 23, 1, 32, 86, 55, 3, 62, 10, 16, 17}

为例，设哈希函数定义为 hash(k) =k % 7，hash(k)对应哈希基表 basetable 的下标值 i。实际上，哈希链表是多条单向链表，哈希基表则是一个结点数组，它的数据元素类型与单向链表的结点类型相同，哈希基表记录多条链表。采用哈希链法的哈希表结构如图 8.8 所示。

基于散列链法的哈希查找操作的过程描述如下：

- 设给定关键字值为 k，计算哈希函数值 i = hash(k)，若哈希基表中位置 i 处的元素 basetable[i]为 null，则查找不成功。
- 否则，检查 basetable[i]的关键字是否等于 k，相等则说明查找成功；不相等则说明产生冲突，需要在由 basetable[i].Next 指向的哈希链表中继续按顺序查找。查找该链表进一步确定查找是否成功。

哈希链表是动态的，同义词越多，链表越长。因此要设计好的哈希函数，使数据尽量均匀地分布在哈希基表中。如果哈希函数的均匀性较差，则会造成哈希基表中空闲单元较多，而某些哈希链表可能很长的情况。一般来说，哈希链表越短越好，而哈希链表过长，则会占用较大的存储空间，降低查找效率。

散列链法克服了试探法的缺陷，无需另外考虑溢出问题，也不会产生堆积现象，而且可以随时对哈希表进行插入、删除和修改等操作。因而散列链法是一种有效的存储结构和查找方法。

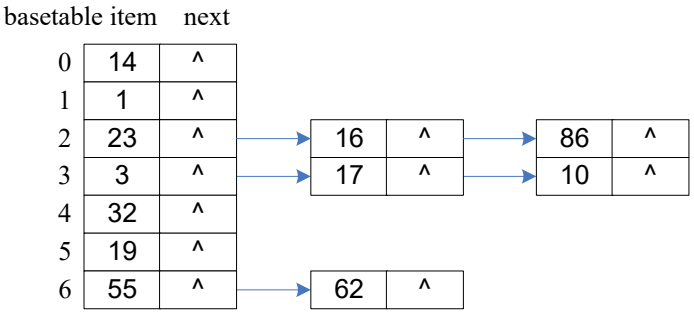

图 8.8 散列链法的哈希表

定义基于散列链法的哈希查找表 HashSearchList 类，它的数据成员 basetable 是一个结点数组，作为哈希基表使用，结点的类型为 Node 类。在哈希查找表 HashSearchList 类中，方法 Hash(k)计算哈希函数值，哈希函数选用 hash(k) = k % p 类型。Add 方法在哈希表中插入结点，Search /Contains 方法在哈希表中查找给定值。

HashSearchList 类和 Node 类的定义如下：

```
public class HashSearchList<T> {
    private Node<T>[] basetable;

    public HashSearchList(int hashsize) {
        basetable = new Node<T>[hashsize];
    }
    public HashSearchList(): this(7) { }

    public int Hash(T k) {
        return k.GetHashCode() % basetable.Length;
```

```
}

public void Add(T k) {
    Node<T> q = new Node<T>(k);
    int i = Hash(k);
    if (basetable[i] == null)
        basetable[i] = q;
    else {
        q.Next = basetable[i].Next;
        basetable[i].Next = q;
    }
}

public Node<T> Search(T k) {
    int i = Hash(k);
    if (basetable[i] == null)
        return null;
    else {
        if (k.Equals(basetable[i].Item))
            return basetable[i];
        else {
            Node<T> q = basetable[i].Next;
            while (q != null && !k.Equals(q.Item)) {
                q = q.Next;
            }
            return q;
        }
    }
}

public bool Contains(T k) {
    Node<T> q = Search(k);
    if (q != null) return true;
    else return false;
}

public void Show() {
    for (int i = 0; i < basetable.Length; i++) {
        Console.Write("Basetable[" + i + "]= ");
        if (basetable[i] != null){
            Node<T> q = basetable[i];
            while (q != null) {
                Console.Write(q.Item + "-> ");
```

```
                    q = q.Next;
                }
            }
            Console.WriteLine(".");
        }
    }
}
```

Node 类的定义如下：

```
public class Node<T> {
    private T item;                    //存放结点值
    private Node<T> next;              //后继结点的引用
    public Node(T k) { item = k; next = null; }
    public T Item { get { return item; } set { item = value; } }
    public Node<T> Next {get { return next; } set { next = value; } }
}
```

【例8.5】　测试哈希查找表建表及查找过程。

HashSearchList 类的测试程序如下所示：

```
using System; using DSAGL;
namespace searchtest {
    class HashSearchListTest {
        static void Main(string[] args) {
            int n = 20; int k = 16;
            int[] d = { 19, 14, 23, 1, 32, 86, 55, 3, 62, 10, 16, 17 };
            HashSearchList<int> hsl1 = new HashSearchList<int>();
            CreateHashList(hsl1, d);
            hsl1.Show();
            Console.WriteLine("hash({0})={1}", k, hsl1.Hash(k));
            Console.WriteLine("Contains({0})={1}", k, hsl1.Contains(k));
        }
        static void CreateHashList(HashSearchList<int> hsl, int[] d) {
            for (int i = 0; i < d.Length; i++)
                hsl.Add(d[i]);
        }
    }
}
```

程序运行结果如下：

```
Basetable[0]= 14-> .
Basetable[1]= 1-> .
Basetable[2]= 23-> 16-> 86-> .
Basetable[3]= 3-> 17-> 10-> .
Basetable[4]= 32-> .
```

```
Basetable[5]= 19-> .
Basetable[6]= 55-> 62-> .
hash(16)=2
Contains(16)=True
```

从查找过程得知，哈希表查找的平均查找长度取决于以下因素：

1) 选用的哈希函数；
2) 选用的处理冲突的方法；
3) 哈希表饱和的程度，常用装载因子 $t=n/m$ 的大小来衡量哈希表饱和的程度，其中 n 为数据元素个数，m 为表的长度。可以证明哈希表的平均查找长度能限定在某个范围内，它是装载因子 t 的函数，而不是数据元素个数 n 的函数，亦即哈希表的查找在常数时间内完成，称其时间复杂度为 $O(1)$。

习题 8

8.1 试分别画出在有序表{1, 2, 3, 4, 5, 6, 7, 8}中查找 6 和 10 的二分查找过程。

8.2 试分别写出对有序表数据进行二分查找的非递归与递归算法实现。

8.3 在一棵空的二叉查找树中依次插入关键字序列{12，7，17，11，16，2，13，9，21，4}，请画出所得到的二叉查找树。

8.4 假设在有 20 个元素的有序数组 a 上进行二分查找，则比较一次查找成功的结点数为 ___；比较两次查找成功的结点数为_____；比较四次查找成功的结点数为 ___；在等概率的情况下查找成功的平均查找长度为_____。设有 100 个结点，用二分查找时，最大比较次数是 ____。设有 22 个结点，当查找失败时，至少需要比较_____次。

8.5 假设在有序表{2, 8, 13, 16, 27, 36, 78}中进行二分查找，请画出判定树，并分别给出查找 16 和 40 时 BinarySearch 方法的返回值。

8.6 哈希查找的设计思想是什么？哈希技术中的关键问题有哪些？

8.7 设哈希表的地址范围为 0～17，哈希函数为：H(k) ＝ k %16。用线性探测法处理冲突，说明对输入关键字序列{10，24，32，17，31，30，46，47，40，63，49}的哈希造表及查找过程。

8.8 设哈希基表的地址范围为 0～9，哈希函数为：H(k) ＝ k %10。用散列链法处理冲突，说明对输入关键字序列{10，24，32，17，31，30，46，47，40，63，49}的哈希造表及查找过程。