



Wuhan University

电子信息学院



武汉大学

数据结构与算法

(C#语言版)

DATA STRUCTURE & ALGORITHM IN C#

第8章 查找算法

王文伟 Wang Wenwei, Dr.-Ing.

Tel: 189-71562600

Email: wwwang@aliyun.com

Web: <http://ipl.whu.edu.cn/sites/ced/st/>

电子信息学院	Table of Contents	武汉大学 Wuhan University
本章位置	第1章 绪论	
	第2章 线性表	
	第3章 栈与队列	
	第4章 串	
	第5章 数组和广义表	
	第6章 树和二叉树	
	第7章 图	
	第8章 查找算法	本章介绍查找操作的基本概念，讨论若干种经典的查找技术；此外还将讨论各种查找算法所适用的数据存储空间结构，以及分析、比较各算法的效率。
	第9章 排序算法	

电子信息学院	Table of Contents	武汉大学 Wuhan University	
8.0 简介			
8.1 查找与查找表			
8.2 线性表的查找			
8.3 二叉查找树及其查找算法			
8.4 哈希查找			

8.0 Introduction


- ◆ **search:** **查找**操作是在数据集中寻找满足某种给定条件的数据元素的过程, **按内容寻找对象**。查找是数据处理中经常使用的一种重要运算, 也是程序设计中的一项重要的基本技术。生活中经常用到查找, 如在字典中查找单词, 在电话簿中查找电话号码等。
- ◆ 本章介绍查找相关的基本概念, 讨论多种经典查找技术, 包括线性表的**顺序**、**折半**和**分块查找**算法, **二叉排序树**的查找算法以及**哈希查找**算法。

4

8.1 查找与查找表

8.1.1 查找操作相关基本概念

8.1.2 C#内建数据结构中的查找操作




第8章 查找算法

5

1. 关键字、查找操作、查找表与查找结果

- ◆ **关键字**：是数据元素类型中用于识别不同元素的某个域（字段）。
 - 若此关键字可以唯一地标识一个记录，则称之为“**主关键字**”。
 - 若此关键字能标识若干记录，则称之为“**次关键字**”。
- ◆ **查找**：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。一般是，根据给定的某个值，在数据集合中确定一个其关键字等于给定值的数据元素（或记录）。
- ◆ **查找表**即被实施查找操作的数据集合，是由同一类型的数据元素(或记录)构成的集合。

第8章 查找算法

6

查找操作与查找结果

- ◆ 查找操作也可以说是按关键字的内容找到数据元素。
- ◆ 若查找表中存在满足条件的数据元素（记录），则称“**查找成功**”，查找结果：给出整个记录的信息，或指示该记录在查找表中的位置；
- ◆ 否则称“**查找不成功**”，查找结果：给出“空记录”或“空指针”。
- ◆ 对查找表经常进行的操作：
 - 查询某个“特定的”数据元素是否在查找表中；
 - 检索某个“特定的”数据元素的各种属性；
 - 在查找表中插入一个数据元素；
 - 从查找表中删去某个数据元素。

2. 静态查找表与动态查找表

- ◆ **静态查找表**（static search table）：
仅作查询和检索操作的查找表，不需要对查找表进行插入、删除操作。例如，字典是一个静态查找表。
- ◆ **动态查找表**（dynamic search table）：
有时在查询之后，还需要将“查询”结果为“不在查找表中”的数据元素**插入**到查找表中；或者，从查找表中**删除**其“查询”结果为“在查找表中”的数据元素。例如，电话簿则是一个动态查找表。

3. 如何进行查找？

- ◆ 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时不得不进行一系列值的比较操作。顺序查找是基本方法，要提高查找效率，需要特定的查找方法。
- ◆ 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。如果数据元素之间不存在明显的组织规律，则不利于查找。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，换句话说，用另外一种结构来表示数据集合。

查找方法

- ◆ 查找的方法与**查找表的规模**的关系：
 - 数据量较小的线性表，可以采用**顺序查找算法**。例如个人电话簿。
 - 数据量较大时，采用**分块查找算法**。例如在字典中查找单词。
- ◆ 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。
- ◆ 衡量查找算法效率的最主要标准是**平均查找长度**（Average Search Length, **ASL**）。**ASL**是指查找过程所需进行的**关键字比较**次数的平均值。

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

p_i 是要查找的数据元素出现在位置*i*处的概率， c_i 是查找相应数据元素需进行的关键字比较次数。考虑概率相等的情况， $p_i=1/n$ 。查找成功和查找不成功的平均查找长度通常不同，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#语言中定义了许多使用方便的数据结构，以Array、List、ArrayList、Hashtable和Dictionary为例介绍其中的查找操作方面的内容。
- ◆ 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**

```
int IndexOf<T>(T[] ar, T k);  
int IndexOf<T>(T[] ar, T k, int  
    startIndex, int count);
```

给定数据在数组指定范围内首次出现的位置

二分查找BinarySearch方法

- ◆ 对于已按关键字值排序的数组，可用更高效的二分查找技术：

```
int BinarySearch<T>(T[] ar, T k);
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**，则返回其索引。如果找不到**k**，则返回一个负数**r**，它的按位补码 $\sim r = -r - 1$ 正好是将**k**插入数组**ar**并保持其排序的正确位置。

{29 36 53 56 70 73 79 79 99 99} 50
 ↑
 i=2, r=-3 ar[~r - 1] < k < ar[~r]

Hashtable与Dictionary

- ◆ Hashtable与Dictionary都是表示<键, 值>对(Key-Value Pair)的集合的类, 这些<键, 值>对根据键的哈希码进行组织。它们的元素可以直接通过键来索引。通过键来检索值的速度非常快。

```
using System; using System.Collections.Generic;
public static void Main() {
    Dictionary<string, string> openWith =
        new Dictionary<string, string>();
    openWith.Add("txt", "notepad.exe");
    openWith.Add("bmp", "paint.exe");
    openWith["doc"] = "winword.exe";
    if (!openWith.ContainsKey("doc")) {
        Console.WriteLine("Key \" doc\" is not found.");
    }
}
```

8.2 线性表查找技术

8.2.1 顺序查找

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。

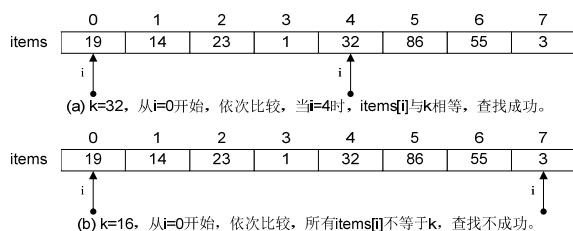
8.2.2 折半查找

8.2.3 分块查找

8.2.1 顺序查找

- ◆ **顺序查找**(sequential search)又称**线性查找**(linear search)，是一种最基本的查找算法。对于给定查找关键字值 k ，从线性表的指定位置开始，依次与每个数据元素的关键字进行比较，直到查找成功，或到达线性表的指定边界时仍没有找到这样的数据元素，则查找不成功。

以顺序表或线性链表表示静态查找表



顺序查找表的定义

- ◆ 设计LinearSearchList类实现顺序表的顺序查找算法。

```
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
    .....
}
```

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）。

顺序查找算法的实现

```
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;  
}
```

单向链表中的查找操作

对于一个给定值 k ，从链表的第一个数据结点沿着链接的方向依次与各结点数据进行比较，直至查找成功或不成功。

```
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;  
}
```

算法分析

- 如果线性表中位置 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。
- 设线性表中元素的个数为 n ，查找第 i 个元素的概率为 p_i ，在等概率情况下， $p_i = 1/n$ ，对于成功的查找，其平均查找长度为：

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

对于不成功的查找，其平均查找长度为

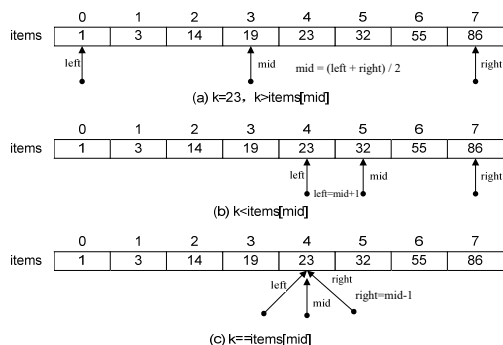
$$ASL_{\text{不成功}} = \sum_{i=0}^{n-1} p_i \times c_i = \sum_{i=0}^{n-1} \frac{1}{n} \times n = n$$

顺序查找算法的时间复杂度为 $O(n)$

8.2.2 二分查找

- 对于有序表（顺序存储结构的数据元素已经按照关键字值的大小排序）可用二分查找(binary search)算法。
- 算法思路：假定元素按升序排列，对于给定值 k ，从表的中间位置开始比较，如果 k 等于当前数据元素的关键字，则查找成功，返回查找到的数据元素的序号。若 k 小于当前数据元素的关键字，则在表的前半部分继续查找；反之，则在表的后半部分继续查找。依次重复进行，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。。

二分查找算法图解



二分查找算法实现

```
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;  
}
```

测试二分查找算法

```
class LinearSearchListTest {
    static void Main(string[] args) { int n = 10;
        var sl=new LinearSearchList<int>(n+8);
        Randomize(sl, n);
        Console.WriteLine("随机排列: "); sl.Show(true);
        Console.WriteLine("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);}
    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

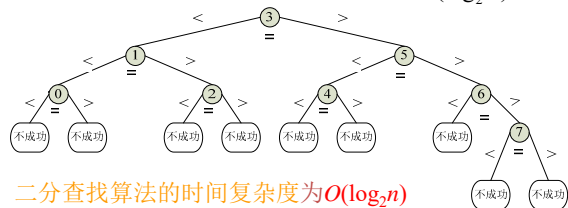
随机排列: LinearSearchList: 73 56 79 53 79 70 99 99 29 36
排序后: LinearSearchList: 29 36 53 56 70 73 79 79 99 99
k=50 re=-3, i= ~re =2

补充说明

二进制:
3的原码: 0000 0011
3的反码: 1111 1100 (反码又称按位补码)
3的补码: 1111 1101 = -3 = -128 + 125
用补码表示负数, 加减法统一计算
补数(十进制): 5 - 3 = 5 + (-3) = 5 + (-10+7) = x 2
补码: 求反加1 反码: 补码减1
2 -> -2(补码) -> -2-1 = -3 = ~2
r = ~2 = -3 i = ~r = -r-1

算法分析

- 二分查找的过程可以用一棵**二叉判定树**表示, 结点中的数字表示数据元素的序号。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。
- 成功与不成功的平均查找长度与n的关系为: $O(\log_2 n)$



二分查找算法的时间复杂度为 $O(\log_2 n)$

8.2.3 分块查找

- 分块查找(blocking search)**将数据存储在若干数据块中, 每一块中的数据顺序存放, 是否排序则不一定, 但多个数据块之间必须按数据的关键字排序(**块间有序**)。假定数据块递增排列, 每个数据块的起始位置记录在另外的一张**索引表**中。通过**索引表**的帮助, 迅速缩小查找的范围。通过索引表的帮助, 对一个数据的查找, 就能限定到一个特定的块中较快地完成。

静态查找表的分块查找

- 静态查找表只需存储、查询, 不需插入、删除。字典
- 字典分块查找算法**的基本思想: 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**, index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单词在dict数组中的起始下标。

index	a	1	dict
	b	48	
	c	102	
	

- 通过索引表, 将较长的单词表dict划分成若干个数据块, 以**首字母相同的若干单词构成一个数据块**, 每个数据块的大小不等, 每块的起始下标由索引表index中对应“首字母”列的“块起始位置下标”列标明。

字典分块查找算法

- 使用**字典分块查找算法**, 在字典dict中查找给定的单词token, 必须分两步进行:
 - 根据token的首字母, 查找索引表index, 确定token应该在dict中的**哪一块**。
 - 在相应数据块中, 使用顺序或折半查找算法查找token, 得到查找成功与否的信息。
- 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, 以限定查找范围。

块内顺序查找

```
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，一般通过在索引表中查找所在块的信息来确定这些参数的值。



动态查找表的分块查找

- ◆ **动态查找表**:需增加或删除数据元素。电话簿
- ◆ 如以顺序存储结构保存数据，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数据，虽然插入、删除操作较方便，但花费的空间较多，查找的效率较低。
- ◆ 以顺序存储结构和链式存储结构相结合的方式存储数据元素，就可能既最大限度地利用空间，又有很高的运行效率。



创建动态分块查找表并测试分块查找算法

- ◆ 定义DynamicLinearBlockSearchList类表示动态分块查找表。对于数据序列：{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}

Block	LinearSearchList
0	6 5 2
1	10
2	23 26
3	33 36
4	43 41 40 46 49
5	57 54 53
6	67 61
7	71 74 72
8	89 80
9	93 92



DynamicLinearBlockSearchList类

```
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];
    }
}
```



插入数据Add(int k)

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



分块查找算法Contains(int k)

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



【例8.3】创建动态查找表，对其测试分块查找

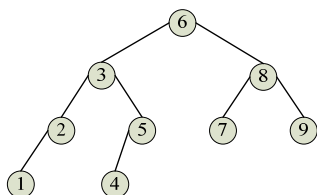
```
class DynamicBlockSearchTest {
    static void Main(string[] args) {
        DynamicLinearBlockSearchList sl = new
            DynamicLinearBlockSearchList(100, 10);
        Randomize(sl, 25); //在查找表中添加25个随机数
        sl.Show();
        bool f = sl.Contains(50);
        Console.WriteLine("Contains("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

8.3 二叉查找树及其查找算法

- ◆ 要提高查找效率 <= 先将数据整理存储。
- ◆ 在普通二叉树中查找，可能需要遍历整棵二叉树，而在“二叉查找树”中查找，仅搜索这种二叉树中的一条路径，不需要遍历整棵树。
- ◆ **二叉查找树(Binary Search Tree, BST)**又称**二叉排序树**，它具有下述性质：
 - 若根结点的左子树非空，则左子树上所有结点的（关键字）值均≤根结点的（关键字）值。
 - 若根结点的右子树非空，则右子树上所有结点的（关键字）值均>根结点的（关键字）值。
 - 根结点的左、右子树也分别为二叉排序树。
- ◆ 二叉排序树的**中根遍历序列是按升序排列的**。

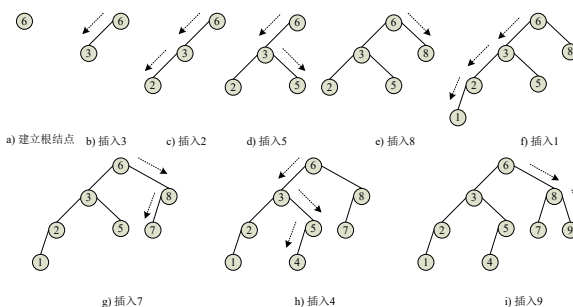
BST及其中根遍历序列

- ◆ 例如序列: {6, 3, 2, 5, 8, 1, 7, 4, 9} 建立的二叉树



- ◆ 它的**中根遍历序列**是{1, 2, 3, 4, 5, 6, 7, 8, 9}

二叉排序树的建立



1) 二叉查找树BinarySearchTree类

- ◆ 定义BinarySearchTree类，它**继承**第六章中定义的二叉树类BinaryTree，结点为BinaryTreeNode类。在BinarySearchTree类中，**Contains**方法在树中查找给定值，**Add**方法在树中插入结点，**构造方法**为给定的数据序列建立二叉查找树。

```
public class BinarySearchTree<T> :
    BinaryTree<T> where T: IComparable{
    public BinarySearchTree(T[] td){
        Console.WriteLine("建立二叉排序树: ");
        for(int i=0; i<td.Length; i++){
            Console.WriteLine(td[i] + " ");
            Add(td[i]);
        }
        Console.WriteLine(); } }
```

2) 在二叉排序树中进行查找

- ◆ 在一棵二叉排序树中查找给定值k的算法描述如下：
 1. 初始化，变量p初始指向根结点。
 2. 进入循环，将k与p结点的关键字进行比较，若两者相等，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，直到查找成功或p为空。
 3. p为非空时表示查找成功，p为空时表示查找不成功。
- ◆ 在二叉查找树查找中，所产生的**比较序列**只是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。也不需要递归算法。

查找算法Contains(T k)

```
public bool Contains(T k) {
    BinaryTreeNode<T> p = root;
    Console.WriteLine("search(" + k + ")= ");
    while (p != null && !k.Equals(p.Data)) {
        Console.WriteLine(p.Data + " ");
        if (k.CompareTo(p.Data) < 0)
            p = p.Left;
        else
            p = p.Right;
    }
    if (p != null)
        return true;
    else return false;
}
```

IPL

第8章 查找算法

43

3) 在二叉查找树中插入结点

1. 如果是空树，则建立数据结点，并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，则进入左子树；如果 k 值较大，则进入右子树。循环迭代，直至当前结点为空结点。
3. 建立数据结点，并将新结点与最后访问的结点进行比较，插到合适的位置。

造树过程，就是将一组数据依次插入二叉排序树中。

IPL

第8章 查找算法

44

插入结点Add(T k)

```
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;
        }
        var t = new BinaryTreeNode<T>(k); //q为最后访问的结点
        if (k.CompareTo(q.Data) <= 0)
            q.Left = t;
        else q.Right = t;
    }
}
```

IPL

第8章 查找算法

45

例8.4: 建立二叉查找树并测试其结果

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

IPL

第8章 查找算法

46

8.4 哈希查找

基本思想

- 前面介绍的查找算法的ASL都与查找表的规模，即表中数据元素的总数有关：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。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。
- 哈希技术是一种按关键字编址的存储和检索数据的方法。哈希(hash)意为杂凑，也称散列。它使用哈希函数(hash function)完成关键字集到地址空间的映射。
- 由数据元素的关键字决定它的存储位置，即哈希函数值 $hash(k)$ ，就是数据 k 的存储位置。按内容查找数据不需要进行多次比较，从而提高了查找的效率。
- 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。

IPL

第8章 查找算法

47

哈希查找技术的关键问题

- 如果关键字 $k_1 \neq k_2$ ，但 $hash(k_1) = hash(k_2)$ ，则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 对应的数据元素称为同义词
- 被处理的数据一般来源于较大的集合，计算机系统地址空间则是有限的，因此哈希函数都是从大集合到小集合的映射，冲突是不可避免的。
- 哈希查找技术的关键问题在于以下两点：
 - 避免冲突(collision avoidance): 设计一个好的哈希函数，尽可能减少冲突。
 - 解决冲突(collision resolution): 发生冲突时，使用一种解决冲突的有效方法。

IPL

第8章 查找算法

48

8.4.2 哈希函数的设计

- ◆ 哈希函数是从大集合（关键字的定义域）到小集合（地址空间）的映射，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，使冲突的机会尽可能地减少。
- ◆ 设计哈希函数，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。
- ◆ 应发挥关键字所有组成成份的作用，充分反映不同关键字之间的差异，这样实现的（关键字到地址的）映射就会比较均匀。

设计哈希函数的几种常用方法

- ◆ **除留余数法**: $\text{hash}(k) = k \% p$
 - ✓ 选 p 为10的某个幂次方
 - ✓ 选 p 为小于哈希表长度的最大素数
- ◆ **平方取中法**: 将关键字值 k^2 的中间几位作为 $\text{hash}(k)$ 的值，位数取决于哈希表的长度。例如， $k = 4731$ ， $k^2 = 22\ 382\ 361$ ，若表长为100，取中间两位，则 $\text{hash}(k) = 82$ 。
- ◆ **折叠法**: 将关键字分成几部分，按照某种规则把这几部分组合在一起。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作为哈希函数值。

Hashtable类使用的哈希函数

- ◆ 不同的查找问题所采用的关键字可能差异很大，每种关键字都有自己的特殊性。不存在一种哈希函数对任何关键字集合都是最好的。在实际应用中，应该构造不同的哈希函数，以求达到最佳效果。
- ◆ 例：Hashtable类使用的哈希函数

```
hash(k) = {k.GetHashCode() + 1 +  
           [ k.GetHashCode() >> 5 + 1 ] % (hashsize - 1) }  
           % hashsize
```

8.4.3 冲突解决方法

- ◆ 冲突不可避免，当冲突发生时必须有效解决冲突。
- ◆ **冲突解决方法**：
 - ◆ 探测定址法（probing rehashing）
 - ◆ 再散列法（rehashing） Hashtable类中采用
 - ◆ 散列链法（hash chaining） Dictionary类中采用

1) 探测定址法（probing rehashing）

- ◆ 基本思想：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 $i = \text{hash}(k)$ ，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。
- ◆ **线性试探法**：欲将数据 k 存放在 $i = \text{hash}(k)$ 位置上。冲突则继续探测下一个空位置。当探测完整个哈希表而没有找到空位置时，说明哈希表已满，再建立一个**溢出表**，原来的哈希表称为哈希基表。

线性试探法

- ◆ 序列：{19, 14, 23, 1, 32, 86, 55, 3, 62, 10}
- ◆ 哈希函数： $\text{hash}(k) = k \% 7$ 造表如下：

哈希基表		溢出表	
0	14	0	3
1	1	1	62
2	23	2	10
3	86	3	
4	32	4	
5	19	5	
6	55	6	

- ◆ **查找时**，首先与**哈希基表**中 $i = \text{hash}(k)$ 位置的数据进行比较，如果该位置是 k ，则查找成功，否则继续向后依次查找。如果在哈希基表中没有找到，还要在**溢出表**中顺序查找。

线性试探法的优缺点

- ◆ 线性试探法是一种较原始的方法，简单，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。

2) 再散列法

Hashtable类中采用

再散列法中定义多个哈希函数：

$$H_i = \text{Hash}_i(\text{key})$$

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) 哈希链表

- ◆ 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。
- ◆ 哈希造表过程：一般将数据元素存放在**哈希基表**的 $i = \text{hash}(k)$ 位置上。如果产生冲突，则创建一个结点存放该数据，并将该结点插入到由冲突的数据元素构成的**哈希链表**。

造表过程举例

- ◆ 序列：{19, 14, 23, 1, 32, 86, 55, 3, 62, 10, 16, 17}
- 哈希函数： $\text{hash}(k) = k \% 7$

baseList item next

0	14	^		
1	1	^		
2	23	^	16	^
3	3	^	17	^
4	32	^		
5	19	^		
6	55	^	62	^

16 ^ 86 ^
17 ^ 10 ^

哈希链表是多条单向链表，哈希基表则是一个结点数组。

哈希链表中的查找

- ◆ 查找 k 时，如果 $\text{baseList}[\text{hash}(k)] == \text{null}$ ，查找不成功。
- ◆ 否则， k 与 $\text{baseList}[\text{hash}(k)]$ 比较，相等则表明查找成功。
- ◆ 不等则表明冲突，在由 $\text{baseList}[\text{hash}(k)].\text{Next}$ 指向的哈希链表中按顺序查找，确定查找是否成功。

baseList item next

0	14	^		
1	1	^		
2	23	^	16	^
3	3	^	17	^
4	32	^		
5	19	^		
6	55	^	62	^

16 ^ 86 ^
17 ^ 10 ^

散列链表的特点

- ◆ 哈希链表是动态的，冲突越多，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。
- ◆ 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。

影响哈希查找技术性能的因素

- ◆选用的哈希函数；
- ◆选用的处理冲突的方法；
- ◆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。已证明哈希表的ASL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 $O(1)$ 。

HashSearchList类的定义

```
public class HashSearchList<T> {
    private Node<T>[] baseList;
    public HashSearchList(int hashsize) {
        baseList = new Node<T>[hashsize];
    }
    public HashSearchList(): this(7) { }
    public int Hash(T k) {
        return k.GetHashCode() % baseList.Length;
    }
    ...Add(); Search(); Show();
}
```

插入数据Add(T k)

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

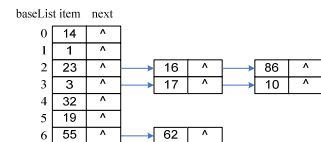
查找数据Search(T, k)

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

【例8.5】测试哈希查找表建表及查找过程

```
using DSAGL;
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> a = new HashSearchList<int>();
        CreateHashList(a, d); a.Show();
        Console.WriteLine("hash({0})={1}", k, a.Hash(k));
        Console.WriteLine("Contains({0})={1}", k, a.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



```
BaseList[0]= 14-> .
BaseList[1]= 1-> .
BaseList[2]= 23-> 16-> 86-> .
BaseList[3]= 3-> 17-> 10-> .
BaseList[4]= 32-> .
BaseList[5]= 19-> .
BaseList[6]= 55-> 62-> .
hash(16)=2
Contains(16)=True
```

本章学习要点

1. 顺序表和有序表的查找方法及其平均查找长度的计算方法。
2. 熟练掌握二叉排序树的构造和查找方法。
3. 熟练掌握哈希表的构造方法，深刻理解哈希表与其它结构的表的实质性的差别。
4. 掌握按定义计算各种查找方法在等概率情况下查找成功时的平均查找长度。