电子信息学院

算法与数据结构
(基于现代C++的方法及实践)
ALGORITHM & DATA STRUCTURE IN MODERN C++

第11章 查找算法

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 189-71562600
Email: wwang@aliyun.com
课程QQ群：珞珈EIS数据结构与算法，668792335

电子信息学院 Table of Contents

本章位置

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结构，以及分析、比较各算法的效率。

电子信息学院 Table of Contents

11.0 Introduction

- search：查找操作是指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，按内容寻找数据对象。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。

11.1 查找与查找表

11.1.1 查找操作相关基本概念
11.1.2 C++内建数据结构中的查找操作

---

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）。
  - 能唯一地标识数据元素的关键字，称为“主关键字”。
  - 若某关键字标识若干而不是唯一元素，称为“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。一般是指根据给定的某个值，在数据集合中找到其关键字等于给定值的数据元素（或记录）。
- 查找表（search table）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。

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）：需要对一个查找表进行插入、删除操作。有时在查询之后，还需要将查询结果为“不在查找表中”的元素插入到查找表中；或者，从查找表中删除查询结果为“在查找表中”的数据元素。例如，电话簿是一个动态查找表。

3. 如何进行查找？(查找方法)

- 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。

查找方法

- 查找表的规模也会影响查找方法的选择：
  - 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操作。
- 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期望值。

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

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。考虑等概率情况。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。

11.1.2 C++内建数据结构中的查找操作

- 数组和C++标准库的vector、list及map等集合类型支持较为方便地实施查找，标准库中的algorithm模块包含了高效、实用的查找算法实现。
- 一般意义下的查找指找到满足一定条件的元素。
- algorithm模块提供多种重载(overloaded)形式的find( )或find_if( )模板函数实现查找操作。[first, last)

```
Iterator find(Iterator first, Iterator last, const T& k);
Iterator find_if(Iterator first, Iterator last,
                 Pred mat);
```

function<bool(const T&)>

```
vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto re = find_if(begin(v), end(v), [](int i){return i%2==0;});
auto s = find_if(begin(t), end(t), [](Student& x){return x.id==9;});
```

推荐设计index函数与其他语言的IndexOf方法一致

- C#/Java的IndexOf方法实现查找操作，返回给定数据在指定范围内首次出现的索引。

```
template <typename IIt, typename T>
int index(IIt first, IIt last, const T& k) {
    // call <algorithm> std::find()
    auto p = find(first, last, k);
    if (p == last) {return -1;  }
    else{ return (int)(p - first);}
}
```

```
int i = index (begin(v), end(v), 10);
```


二分查找binary_search函数

- 对于已按关键字值排序的集合，可用更高效的二分查找技术：

bool binary_search(It first, It last, const T& k);
在范围[first, last)中应用二分查找算法查找具有指定值k的元素，如果找到，则返回 true；否则，返回 false。

int BinarySearch<T>(T[] ar, T k);　　C#/Java
返回给定数据k在数组ar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，它的反码 ~r =i= -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
r=-3 => i=2　　ar[~r - 1] < k < ar[~r]
i-1　　i

C#语言集成了高级的LINQ查询模块

```
IEnumerable<Robot> qv =
   from r in robots
      where r.Name.Contains(textBox1.Text)
      select r;

foreach(var r in qv) {
  listBox1.Items.Add("名字：" + r.Name+
    " ID：" + r.ID + "  智商：" + r.IQ);
}
```


C++标准库中的关联容器map

- 关联容器是表示<键，值>对（Key-Value Pair）的容器类，这些<键，值>对根据键的哈希码进行组织。提供了从一组键到一组值的映射，键的作用类似于数组中的下标，可以通过键来索引集合内的元素。通过键来检索值的速度非常快，时间效率接近于O(1)。
- 在C#、Java和Python等编程语言中，map这种类型称为字典Dictionary。

11.2 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。

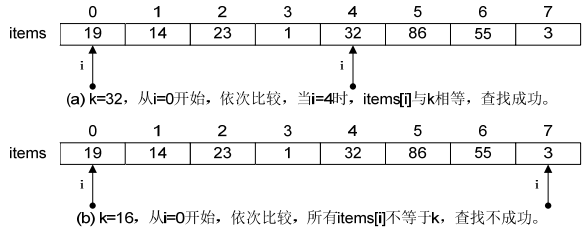
11.2.1 顺序查找
11.2.2 折半查找
11.2.3 分块查找

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。

11.2.1 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。

数组和线性表是典型的顺序查找表



顺序查找表的定义

- 设计模块LinearSearch实现顺序查找算法。

```
template <typename IIt, typename T>
int Index(IIt first, IIt last, const T& k) {
  int i = 0; auto pitems = first;
  while(pitems<last&&*pitems!=k){
    i++; pitems++;
  }
  if (pitems == last)return -1;
  else return i;
}
```


顺序查找算法的实现（II）

```
template <typename IIt, typename T>
bool Contains(IIt first, IIt last, const T& k) {
  int j = Index(first, last, k);
  if (j != -1) return true;
  else  return false;}
template <typename IIt, typename Predicate>
int IndexIf(IIt first,IIt last,Predicate pred) {
  int i = 0; auto pitems = first;
  while (pitems < last && !pred(*pitems)) {
    i++; pitems++;}
  if (pitems == last)return -1;
  else return i; }
```


应用举例

在一个双向链表中查找第一个偶数的序号，返回值表达查找的结果。容易看出，该函数也能方便地应用于数组、vector等数据结构。

```
list<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9};
int re= IndexIf(begin(v),end(v),[](int i)
       {return i%2==0;});
```


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。c_i=i+1
- 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：

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

对于不成功的查找，其平均查找长度为

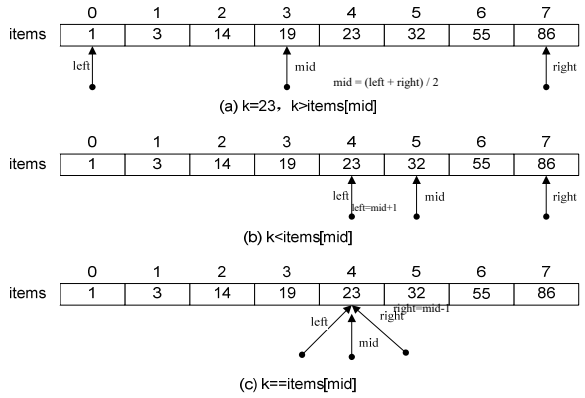
$$ASL_{不成功}=\sum_{i=0}^{n-1}p_i\times c_i=\sum_{i=0}^{n-1}\frac{1}{n}\times n=n$$

顺序查找算法的时间复杂度为$O(n)$

11.2.2 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。
- 查找不成功，应该返回一个有意义的负数。

二分查找算法图解



二分查找算法实现

```
template <typename IIt, typename T>
int BinarySearch(IIt first, IIt last, const T& k) {
  auto items = first; int length = (int)(last - first);
  int mid = 0, left = 0; int right = length - 1;
  while (left <= right) {
    mid = (left + right) / 2;
    if (k == items[mid]) {return mid;}
    else if (k < items[mid]) right = mid - 1;
    else left = mid + 1;
  }
  if (k > items[mid]) mid++;
  return ~mid;
}
```


测试二分查找算法

```
#include "../dsa/LinearSearch.h"
#include "../dsa/dsaUtils.h"
int main() { // LinearSearchTest.cpp
  const int CNT = 12; int items[CNT];
  RandomizeData(items,CNT,7,1,100); // seed=7, [1,100)
  cout << "随机排列: "; Show(items, CNT);
  cout << "排序后 : ";
  sort(items, items + CNT); Show(items, CNT);
  int k = 50;
  int re = Index(items, items + CNT, k);
  cout<<re<< endl;
  re = BinarySearch(items, items + CNT, k);
  cout << "k= " << k << ", re= " << re <<
          ", i=~re= " << ~re << endl;    }
```


程序运行结果

```
随机排列: 1 53 46 22 47 20 92 36 84 62 15 86
排序后 : 1 15 20 22 36 46 47 53 62 84 86 92
-1                                  k= 50, re= -8, i=~re= 7
```

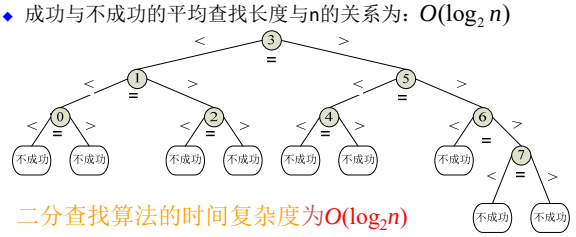
补充说明

二进制:
3的原码: 0 0 0 0 0 0 1 1
3的反码: 1 1 1 1 1 1 0 0 (反码又称按位补码)
-3用3的补码表示: 1 1 1 1 1 1 0 1 = -128 + 125 = -3
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 (反码) = -2-1= -3 (反码)
r = -3 => i = ~r = -r-1 = 3-1 = 2

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$O(\log_2 n)$

11.2.3 分块查找

- 当数据量较大时，顺序查找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。
- 分块查找将数据存储在若干数据块中，每一块中的数据顺序存放。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，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，迅速缩小查找的范围 。

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
b	48
c	102
…	….

dict { A块 / B块 }

- 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

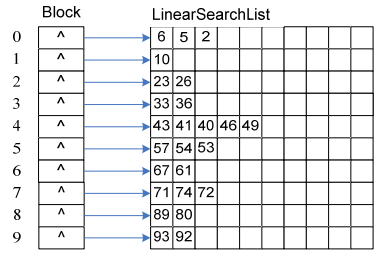
- 使用分块查找算法，在字典dict中查找给定的单词token，必须分两步进行：
  1. 根据token的首字母，查找索引表index，确定token应该在dict中的哪一块。
  2. 在相应数据块中，使用顺序或折半查找算法查找token，得到查找成功与否的信息。
- 在某一数据块内进行顺序查找，可以通过函数Index来完成，给它提供参数来限定查找范围。

动态查找表的分块查找

- 动态查找表：除查找外，常常还需增加或删除数据元素。如，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。

创建动态分块查找表并测试分块查找算法

- 定义BlockSearch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}



BlockSearch类

```
class BlockSearch{
private: SequencedList<int>* *_blocks;
int _blocksize, _blocknum;
public:
  BlockSearch(int capacity=100, int blocksize=10) {
  _blocksize = blocksize;
  if(capacity%blocksize==0)_blocknum=capacity/ blocksize;
  else _blocknum = capacity/blocksize + 1;
  _blocks = new SequencedList<int>*[_blocknum];
  for(int i=0; i<_blocknum;i++)
    _blocks[i] = new SequencedList<int>(_blocksize);
}
~BlockSearch(){
   for(int i=0;i<_blocknum;i++)delete _blocks[i];
    delete [] _blocks;     }
```

插入数据insert(int k)

```
void insert(int k) {
  int i = k / _blocksize; if(k<0)i= 0;
  if (k >= _blocksize * _blocknum)
    i = _blocknum - 1;
  _blocks[i]->push_back(k);
}
void insert_range(const int* pdata,int cnt) {
  for(int i=0; i<cnt; i++)
    insert(pdata[i]);
}
```

分块查找算法contains(int k)

```
bool contains(int k) {
  int i = k / _blocksize;
  cout << "search k= " << k <<
       " in Block[" << i <<"]\t";
  bool found = _blocks[i]->contains(k);
          return found;
     }
```


【例11.2】创建动态查找表，对其测试分块查找

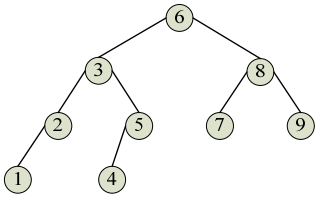
```
#include "../dsa/BlockSearch.h"
#include "../dsa/dsaUtils.h"
int main() { // BlockSearchTest.cpp
  const int CNT = 25; int items[CNT];
  RandomizeData(items, CNT, 9, 1,100);
  // seed=9，25个随机数[1,100)
  Show(items, CNT);
  BlockSearch bslist(100,10);
  bslist.insert_range(items, CNT); bslist.show();
  int k = 50;
  bool f = bslist.contains(k);
  cout << "contains(" << k << ") = " << f;  }
```


11.3 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。

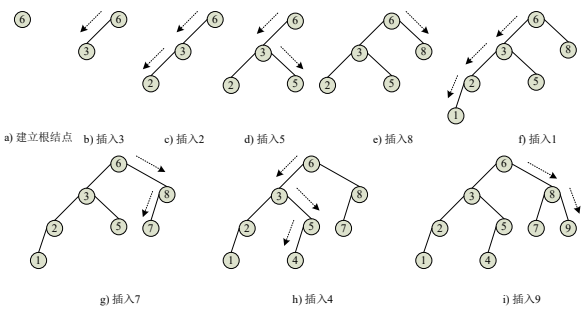
BST及其中根遍历序列

- 例如：以序列{6, 3, 2, 5, 8, 1, 7, 4, 9}建立的二叉查找树



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



1）二叉查找树BSTree类

- 定义BSTree类，它继承BTree类，结点为BTNode类。在BSTree类中，contains和或search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。

```
template <typename T>
class BSTree : public BTree<T>{
public:
    BSTree() {} // 可以不要
    // ~BSTree() { this->dispose(); }
```

2）在二叉排序树中进行查找

- 在一棵BST中查找给定值*k*的算法描述如下：
  1. 初始化，变量p初始指向树的根结点root。
  2. 进入循环，将*k*与p结点的内容进行比较，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. 退出循环后，p为非空时表示查找成功，p为空时表示查找不成功。
- 在BST查找过程中，所产生的比较序列只是BST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(k)

```
bool contains(const T& k) {
  BTNode<T>* p = this->_root;
  cout << "search(" << k << ")=  ";
  while(p!=nullptr&&k!=p->data){ //比较是否相等
    cout << p->data << " ";
    if (k < p->data) //比较大小
      p = p->left; //进入左子树
    else p=p->right; //进入右子树
  }
  if(p!=nullptr) return true; //查找成功
  else return false;      //查找不成功
}
```


3）在二叉查找树中插入数据k

1. 如果是空树，则为数据k建立一个新结点，并作为BST的根结点。
2. 否则从根结点开始，将数据*k*与当前结点的内容进行比较，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. 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比较，插到合适的位置。

插入蕴含查找

插入结点insert(T k)

造树过程，就是将一组数据依次插入BST树中。

```
void insert(const T& k) {
  BTNode<T> *p, *q=nullptr;
  if (this->_root == nullptr)
    this->_root= new BTNode<T>(k);//建立根结点
  else { p = this->_root;
    while (p != nullptr) {
      q = p;
      if (k <= p->data) p= p->left;
      else p = p->right;  }
    p = new BTNode<T>(k);
    if ( k <= q->data) q->left = p;
    else q->right = p; }  }
```

例11.3：建立二叉查找树并测试其结果

```
#include "../dsa/BSTree.h"
int main() { // BSTreeTest.cpp
  const int CNT = 9;
  int td[CNT] = { 5, 8, 3, 2, 4, 7, 9, 1, 5 };
  BSTree<int> bst;
  bst.insert_range(td, CNT);
  bst.showInOrder();
  bst.dispose();
  return 0; }
```

程序运行结果如下：

```
建立二叉排序树： 583247915
中根次序： 123455789
```


11.4 哈希查找

基本思想

- 前面介绍的算法，它们的平均查找长度都与查找表的规模有关：元素越多，为查找而进行的比较次数就越多。在这些查找表中，数据元素所占据的存储位置与其内容本身无关，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。
- 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）完成关键字值到地址的映射。

数据存储在其哈希函数值指示的位置

- 由数据元素的关键字决定它的存储位置，即将数据k存储在其哈希函数值hash(k)指示的位置。按内容k查找数据，也是直接到位置i=Hash(k)处查找，不需要进行多次比较，从而提高查找的效率。
- 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。

哈希查找技术的关键问题

- 如果关键字k_1!=k_2，但Hash(k_1)=Hash(k_2)，则表示不同关键字数据映射到同一存储位置。此现象称为冲突(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（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。
- 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策略。

11.4.2 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映射到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: Hash(k) = k % p
  - ✓ 选p为10的某个幂次方，如p = 10^3
  - ✓ 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Hash(k)的值，位数取决于哈希表的长度。例如，k = 381，k^2 = 14<u>51</u><u>61</u>，若表长为100，取中间两位，则hash(k) = 51。
- 折叠法:将关键字分成几部分，按照某种规则把这几部分折叠组合在一起。例如移位折叠法，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。

举例：C#的Hashtable类使用的哈希函数

- 不同的查找问题所采用的关键字差异可能很大，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
```

C++等编程语言也有较多应用折叠法的案例，其哈希函数采用与上例相同的构造方式，即先将关键字k转换为无符号整数，接着将整数的不同部分，例如，高位和低位，折叠。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。

11.4.3 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） 应用更多

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* = 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* = Hash(*k*)位置上。产生冲突时继续**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：Hash(k)=k%7　造表如下：

哈希基表

0	14
1	1
2	23
3	86
4	32
5	19
6	55

溢出表

0	3
1	62
2	10
3	
4	
5	
6	

- 查找时，首先与哈希基表中 *i* = Hash(*k*) 位置的数据进行比较，如果该位置的值是*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：

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

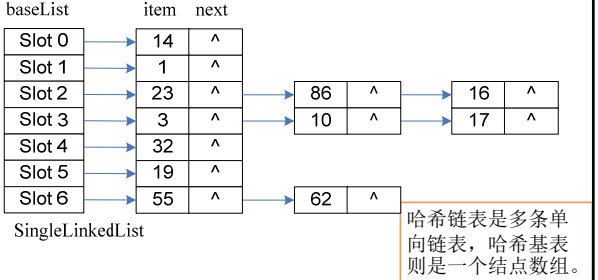
当同义词对一个哈希函数产生冲突时，计算另一个哈希函数，直至冲突不再发生。这种方法不易产生堆积现象，但增加了计算时间。

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，被存储到一个称为哈希链表的线性链表中，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。基表元素又称为哈希槽（hash slot）。
- 哈希造表过程：对于关键字*k*，首先计算其哈希函数值i = Hash(k)，将该数据元素插入到哈希基表位置i处记录的哈希链表baseList[i]中。

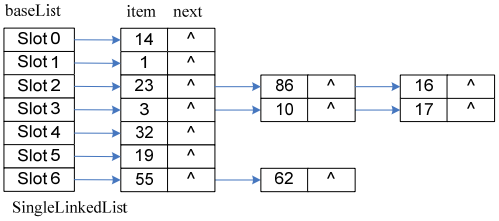
造表过程举例

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



哈希链法中的查找

- 查找*k*时，计算*i*=Hash(*k*)，如果baseList[i]指向的链表为空表，则查找不成功。
- 否则，需要在由baseList[i]指向的哈希链表中继续按顺序查找，确定查找是否成功。



散列链法的特点

- 哈希链表是动态的，产生冲突的同义词越多，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。
- 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。

哈希查找表设计：HashList类的定义

```
#include "SingleLinkedList.h"
class HashList { private:
    SLinkedList<int>** _baseList; int _length;
public:
    HashList(int listsize=10):_length(listsize){
      _baseList = new SLinkedList<int>*[_length];
      for (int i = 0; i < _length; i++)
        _baseList[i] = new SLinkedList<int>(); }
    ~HashList() {
      for(int i=0;i<_length;i++)delete _baseList[i];
      delete[]_baseList;    }
    int Hash(int k) const { return k % _length;}
…insert(); search(); show();  };
```

类设计遵循RAII原则

- 构造函数：哈希基表（_baseList数组）及各空链表被构造出来。
- 析构函数：销毁各链表及哈希基表本身。
- 遵循RAII原则的程序设计实践。

```
HashList(int listsize=10):_length(listsize){
  _baseList = new SLinkedList<int>*[_length];
  for (int i = 0; i < _length; i++)
    _baseList[i] = new SLinkedList<int>(); }
~HashList() {
  for(int i=0;i<_length;i++)delete _baseList[i];
  delete[] _baseList; }
```


插入数据inser(k)和insert_range()

```
void insert(int k) {
  int i = Hash(k);
  _baseList[i]->push_back(k);
  // _baseList[i]->insert(0,k);
}
```

```
void insert_range(const int* pdata, int cnt) {
  int i = 0;
  for (int j = 0; j < cnt; j++) {
    i = Hash(pdata[j]);
    _baseList[i]->push_back(pdata[j]);;
  } }
```


查找数据search(k)

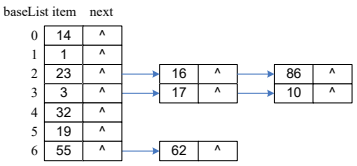
```
const SLNode<int>* search(int k) const {
  int i = Hash(k);
  return _baseList[i]->search(k);
}
```

```
bool contains(int k) const {
  SLNode<int>* q = (SLNode<int>*)search(k);
  if (q != nullptr)
    return true;
  else
    return false;
}
```

【例11.4】测试哈希查找表建表及查找过程

```
#include "../dsa/HashList.h"
#include "../dsa/dsaUtils.h"
int main(int argc, char* argv[]) { // HashListTest.cpp
  const int CNT = 12;
  int d[CNT]{ 19, 14, 23, 1, 32, 86, 55, 3, 62, 10, 16, 17 };
  Show(d, CNT);int k = 16;  HashList hlist(7);
  hlist.insert_range(d, CNT); hlist.show();
  cout<<"hash(" << k << ")= "<< hlist.Hash(k) << endl;
  cout<<"Contains("<<k<<") = "<<hlist.contains(k)<<endl;
  hlist.remove(k); hlist.show();
  cout<<"hash("<<k<< ") = " << hlist.Hash(k) << endl;
  cout<<"Contains("<<k<<") = "<<hlist.contains(k)<<endl;
}
```


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
```


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$O(1)$。

- map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，时间效率接近于O(1)。d[“张三”]

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查找表的实质性差别。
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及不成功时的平均查找长度ASL。