Chapter 5 Hashing

South China University of Technology

College of Software Engineering

Huang Min

The Need for Speed

- Data structures we have looked at so far
  - › Use comparison operations to find items
  - › Need O(log N) time for Find and Insert
- In real world applications, N is typically between 100 and 100,000 (or more)
  - › log N is between 6.6 and 16.6
- Hash tables are an abstract data type designed for **O(1)** Find and Inserts

Fewer Functions Faster

- compare lists and stacks
  - by reducing the flexibility of what we are allowed to do, we can increase the performance of the remaining operations
  - insert(L,X) into a list versus push(S,X) onto a stack
- compare trees and hash tables
  - trees provide for known ordering of all elements
  - hash tables just let you (quickly) find an element

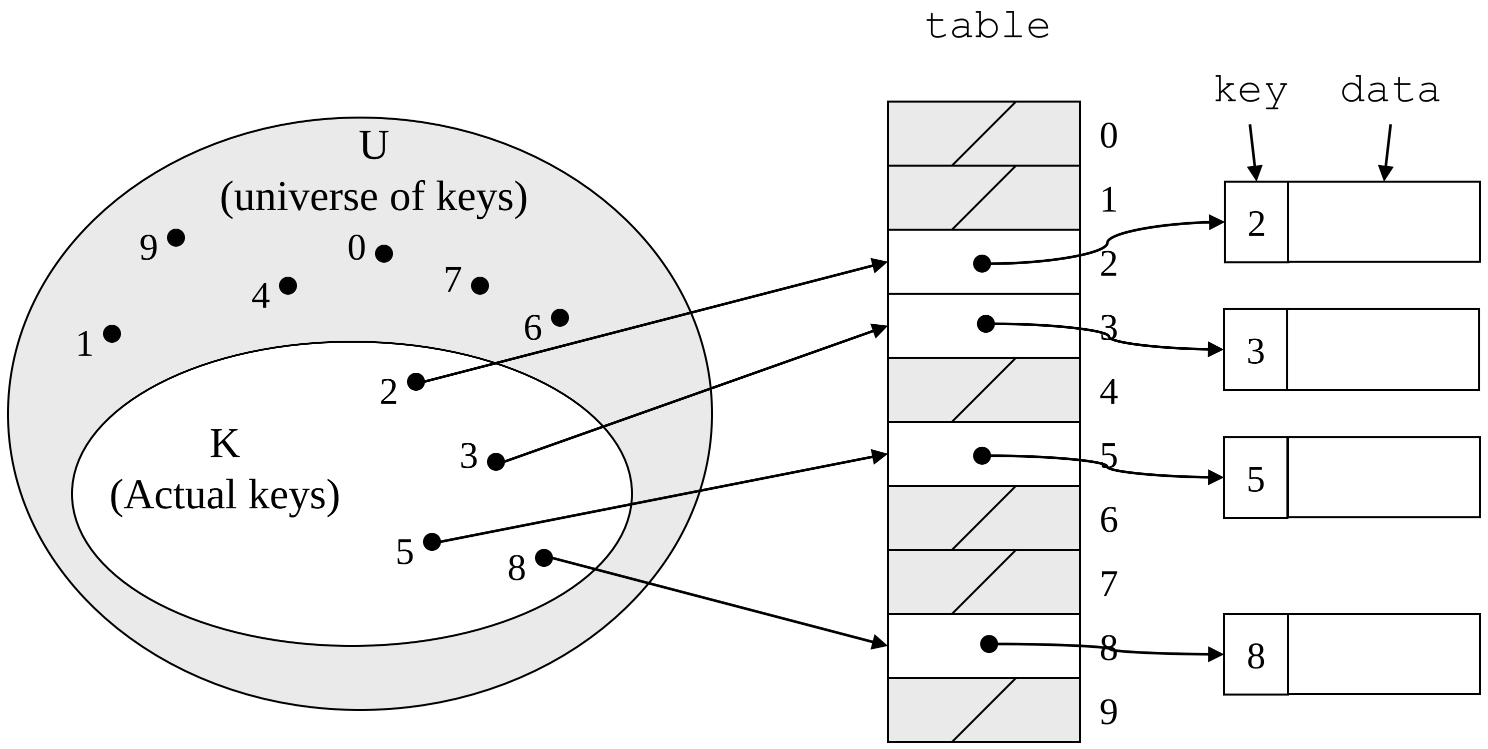
Limited Set of Hash Operations

- For many applications, a limited set of operations is all that is needed
  - Insert, Find, and Delete
  - Note that no ordering of elements is implied

Direct Address Tables

- Direct addressing using an array is very fast
- Assume
  - keys are integers in the set U={0,1,…*m*-1}
  - *m* is small
  - no two elements have the same key
- Then just store each element at the array location array[key]
  - search, insert, and delete are trivial

Direct Access Table



Direct Address Implementation

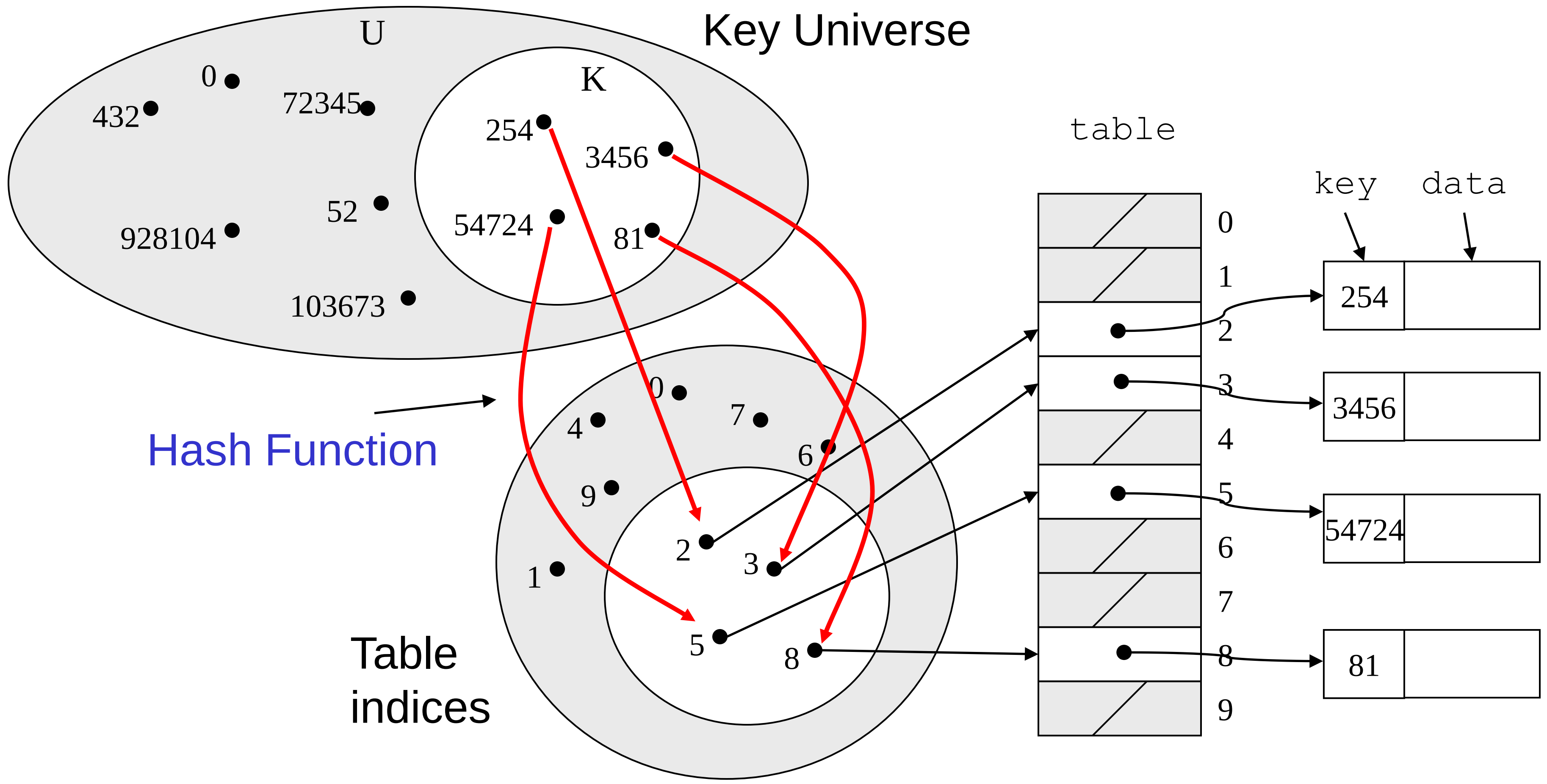
```
Delete(Table T, ElementType x)
   T[key[x]] = NULL    //key[x] is an
                      //integer
Insert(Table t, ElementType x)
   T[key[x]] = x

Find(Table t, Key k)
   return T[k]
```

An Issue

- If most keys in U are used
  - › direct addressing can work very well (m small)
- The largest possible key in U , say m, may be much larger than the number of elements actually stored (|U| much greater than |K|)
  - › the table is very sparse and wastes space
  - › in worst case, table too large to have in memory
- If most keys in U are not used
  - › need to map U to a smaller set closer in size to K

Mapping the Keys



Hashing

Hashing: The process of mapping a key value to a position in a table.

A hash function maps key values to positions. It is denoted by ***h***.

A hash table is an array that holds the records. It is denoted by **HT**.

HT has *M* slots, indexed form 0 to *M*-1.

Hashing

For any value K in the key range and some hash function $\boldsymbol{h}$, $\boldsymbol{h}(K) = i$, $0 <= i < M$, such that **HT**[i] = K.

Hashing is appropriate only for sets (no duplicates).

Good for both in-memory and disk-based applications.

Hashing is not good for range queries, minimum or maximum queries and queries in key order.

Collisions

Given: hash function *h* with keys k_1 and k_2. β is a slot in the hash table.

If $h(k_1) = \beta = h(k_2)$, then k_1 and k_2 have a collision at β under *h*.

So, search for the record with key *K*:

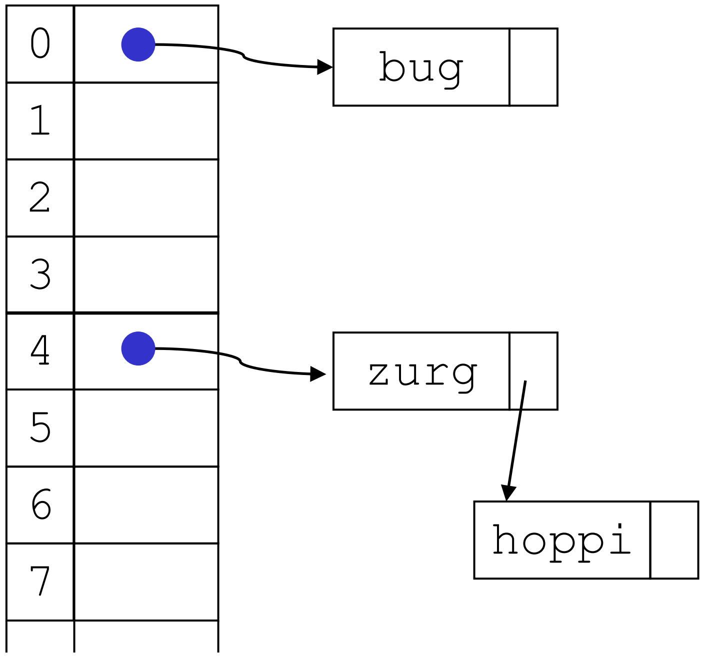
1. Compute the table location ***h***(*K*).
2. Starting with slot ***h***(*K*), locate the record containing key *K* using (if necessary) a collision resolution policy.

Collision Resolution

- Separate Chaining
  - › Use data structure (such as a linked list) to store multiple items that hash to the same slot
- Open addressing (or probing)
  - › search for empty slots using a second function and store item in first empty slot that is found

Resolution by Chaining

- Each hash table cell holds pointer to linked list of records with same hash value
- Collision: Insert item into linked list
- To Find an item: compute hash value, then do Find on linked list
- Note that there are potentially as many as TableSize lists



Resolution by Open Addressing

- No links, all keys are in the table
  - reduced overhead saves space
- When searching for `X`, check locations $h_1(X)$, $h_2(X)$, $h_3(X)$, … until either
  - `X` is found; or
  - we find an empty location (`X` not present)
- Various flavors of open addressing differ in which probe sequence they use

Cell Full? Keep Looking.

- **`h`$_i$`(X)=(Hash(X)+F(i)) mod TableSize`**
  - Define F(0) = 0
- F is the collision resolution function. Some possibilities:
  - Linear: F(i) = i
  - Quadratic: F(i) = i^2
  - Double Hashing: F(i) = $i \cdot \text{Hash}_2(X)$

Linear Probing

- When searching for `K`, check locations `h(K)`, `h(K)+1`, `h(K)+2`, … mod TableSize until either
  - › `K` is found; or
  - › we find an empty location (`K` not present)
- If table is very sparse, almost like separate chaining.
- When table starts filling, we get clustering but still constant average search time.
- Full table $\Rightarrow$ infinite loop.

Linear Probing Example

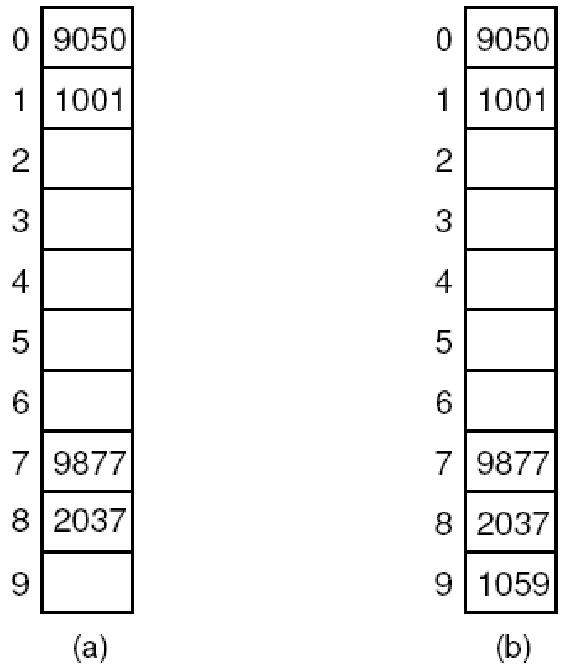


Figure 9.8 Example of problems with linear probing. (a) Four values are inserted in the order 1001, 9050, 9877, and 2037 using hash function $\mathbf{h}(K) = K \bmod 10$. (b) The value 1059 is added to the hash table.

Primary Clustering Problem

- Once a block of a few contiguous occupied positions emerges in table, it becomes a “target” for subsequent collisions
- As clusters grow, they also merge to form larger clusters.
- Primary clustering: elements that hash to different cells probe same alternative cells

Quadratic Probing

- When searching for **`X`**, check locations $h_1(X)$, $h_1(X) + 1^2$, $h_1(X)+2^2$,… **`mod TableSize`** until either
  - **`X`** is found; or
  - we find an empty location (**`X`** not present)

Example: M=101

- $\boldsymbol{h}(k_1)$=30, $\boldsymbol{h}(k_2)$ = 29.
- Probe sequence for k_1 :30， 31， 34， 39
- Probe sequence for k_2 :29， 30， 33， 38

Disadvantage: not all hash table slots will necessarily be on the probe sequence.

Secondary Clustering

Quadratic probing eliminate primary clustering—the problem of keys sharing substantial segments of a probe sequence.

But if two keys hash to the same home position, they follow the same probe sequence. This is called secondary clustering.

To avoid secondary clustering, need probe sequence to be a function of the original key value, not just the home position.

Double Hashing

- When searching for `X`, check locations `h`$_1$`(X)`, `h`$_1$`(X)+ h`$_2$`(X)`,`h`$_1$`(X)+2*h`$_2$`(X)`,… `mod Tablesize` until either
  - `X` is found; or we find an empty location (`X` not present)

Probe sequences is $(\boldsymbol{h}(k)+i^* \boldsymbol{h}_2(K))\%M$ (i>=0)

Example: Hash table of size M=101

- $\boldsymbol{h}(k_1)=30$, $\boldsymbol{h}(k_2)=28$, $\boldsymbol{h}(k_3)=30$.
- $\boldsymbol{h}_2(k_1)=2$, $\boldsymbol{h}_2(k_2)=5$, $\boldsymbol{h}_2(k_3)=5$.
- Probe sequence for k_1 is: 30, 32, 34, 36.
- Probe sequence for k_2 is: 28, 33, 38, 43.
- Probe sequence for k_3 is: 30, 35, 40, 45.

Load Factor of a Hash Table

- Let N = number of items to be stored
- Load factor λ = N/TableSize
  - TableSize = 101 and N =505, then $\lambda = 5$
  - TableSize = 101 and N = 10, then $\lambda = 0.1$

Deletion

Two important considerations for deleting records from a hash table:

1) Deleting a record must not hinder later searches.

2) We do not want to make positions in the hash table unusable because of deletion.

Tombstones

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a tombstone, which indicates that a record once occupied the slot but does so no longer.

A tombstone will not stop a search, but that slot can be used for future insertions.

Tombstones

Unfortunately, tombstones add to the average path length.

Solutions:

1. Local reorganizations to try to shorten the average path length. For example, after deleting a key, continue to follow the probe sequence of that key and swap records further down the probe sequence into the slot of the recently deleted record .

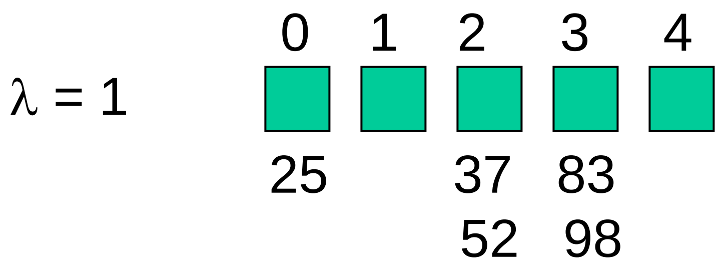
2. Periodically rehash the table (by order of most frequently accessed record).

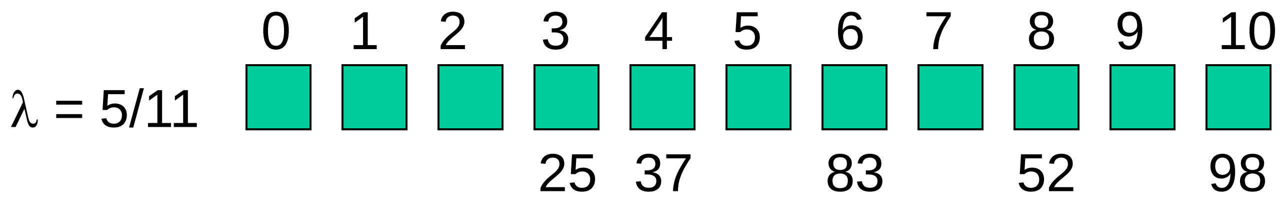
Rehashing

- Build a bigger hash table of approximately twice the size when λ exceeds a particular value
  - Go through old hash table, ignoring items marked deleted
  - Recompute hash value for each non-deleted key and put the item in new position in new table
  - Cannot just copy data from old table because the bigger table has a new hash function

Rehashing Example

- Open hashing – $h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$.





散列(Hashing)基本概念

散列表 (Hash table，也叫哈希表)，它是基于快速存取的角度设计的，也是一种典型的“空间换时间”的做法。顾名思义，该数据结构可以理解为一个线性表，但是其中的元素不是紧密排列的，而是可能存在空隙。

数组的特点是：寻址容易，插入和删除困难；

链表的特点是：寻址困难，插入和删除容易；

Hash表是一种能综合两者优点、既寻址容易又插入和删除容易的数据结构。

散列表是根据关键字(key value)而直接进行访问的数据结构，它通过把key值映射到表中的一个位置来访问记录，从而加快查找的速度。

- 实现key值映射的函数就叫做散列函数。
- 存放记录的数组就就叫做散列表。
- 实现散列表的过程通常就称为散列(hashing)

hash函数的选择 (1)

- 理论上，对任意一类的数据存在一个完美的哈希函数（没有任何碰撞的、不出现重复的散列值），但现实中很难找到，而且这种完美函数的趋近变种在实际应用中的作用相当有限。

- 实践中的完美哈希函数是指在一个特定的数据集上产生的碰撞最少的哈希函数。

hash函数的选择 (2)

- 可以从下面两个角度来选择哈希函数：

1. 数据分布

　　一个衡量的标准是考虑一个哈希函数是否能将一组数据的哈希值进行很好的分布。

　　可以通过碰撞的哈希值的个数来进行分析。如果用链表来处理碰撞，则可以分析链表的平均长。

2. 哈希函数的效率

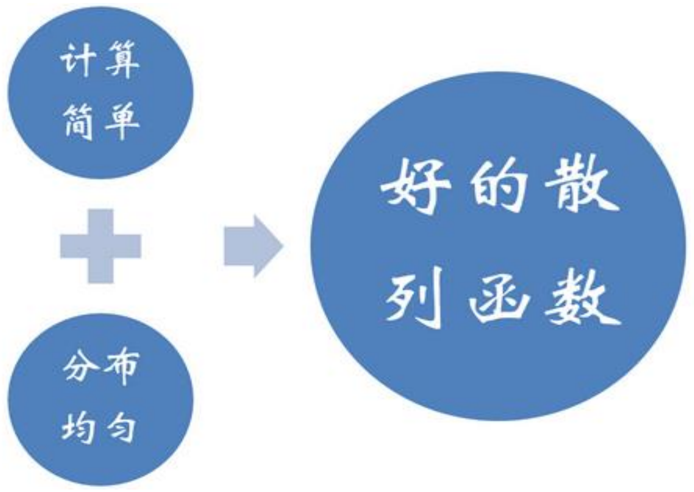
　　另个一个衡量标准是哈希函数得到哈希值的效率。

　　通常哈希函数的算法复杂度都假设为O(1)，而另外一些常用的数据结构，比如树则被认为是O(logn)的复杂度。

　　一个好的哈希函数必须在理论上非常的快、稳定并且是可确定的。通常很多哈希函数不可能达到O(1)的复杂度。

散列函数的构造 (1)

- **构造散列函数的两个基本原则**



计算简单：指散列算法简单快捷，散列值生成简单。

分布均匀：指对于key值集合中的任一关键字，散列函数能够以均匀的概率映射到数组的任意一个索引位置上，这样能够减少散列碰撞。

散列函数的构造 (2)

1. 直接寻址法：取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a*key + b，其中a和b为常数（这种散列函数叫做自身函数）

2. 数字分析法：分析一组数据，比如一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体相同，这样的话，出现冲突的几率就会很大，但是我们发现年月日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成散列地址，则冲突的几率会明显降低。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址。

3. 平方取中法：取关键字平方后的中间几位作为散列地址。

散列函数的构造 (3)

4. 折叠法：将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。

5. 随机数法：选择一随机函数，取关键字的随机值作为散列地址，通常用于关键字长度不同的场合。

6. 除留余数法：取关键字被某个不大于散列表表长m的数p除后所得的余数为散列地址。即 H(key) = key MOD p, p<=m。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对p的选择很重要，一般取素数或m，若p选的不好，容易产生同义词（具有相同函数值的关键字对该哈希函数来说称为同义词）。

hash冲突及解决

- hash冲突在所难免，解决冲突是一个复杂问题。
- 冲突主要取决于：
  （1）与散列函数有关，一个好的散列函数的值应尽可能平均分布。
  （2）与解决冲突的哈希冲突策略有关。
  （3）与负载因子的大小有关，太大太小都不好，太大容易产生冲突和聚集，而太小浪费空间严重。

- 解决冲突的办法：
  （1）开散列法：把所有同义词，即hash值相同的记录，用单链表连接起来。
  （2）闭散列法：线性探查法、平方探查法、伪随机序列法、双哈希函数法。

哈希(Hash)与加密(Encrypt) (1)

- **哈希与加密的区别**
  1. 哈希算法往往被设计成生成具有相同长度的文本，而加密算法生成的文本长度与明文本身的长度有关。

  例如有两段文本：“Microsoft”和“Google”，使用某种哈希算法得到的结果分别为：“140864078AECA1C7C35B4BEB33C53C34”和“8B36E9207C24C 76E6719268E49201D94”。而使用某种加密算法的结果分别为“Njdsptpgu”和“Hpphmf”。哈希的结果具有相同长度，而加密的结果则长度不同。

  实际上，如果使用相同的哈希算法，不论输入有多么长，得到的结果长度是一个常数，而加密算法往往与明文的长度成正比。

哈希(Hash)与加密(Encrypt)（2)

- **哈希与加密的区别**

  2、哈希算法是不可逆的，而加密算法是可逆的。

  这里的不可逆有两层含义，一是“给定一个哈希结果R，没有方法将R转换成原目标文本S”，二是“给定哈希结果R，即使知道一段文本S的哈希结果为R，也不能断言当初的目标文本就是S”。

  哈希是不可能可逆的，因为如果可逆，那么哈希就是世界上最强悍的压缩方式了——能将任意大小的文件压缩成固定大小。

  加密则不同，给定加密后的密文R，存在一种方法可以将R确定的转换为加密前的明文S。

哈希(Hash)与加密(Encrypt)（3)

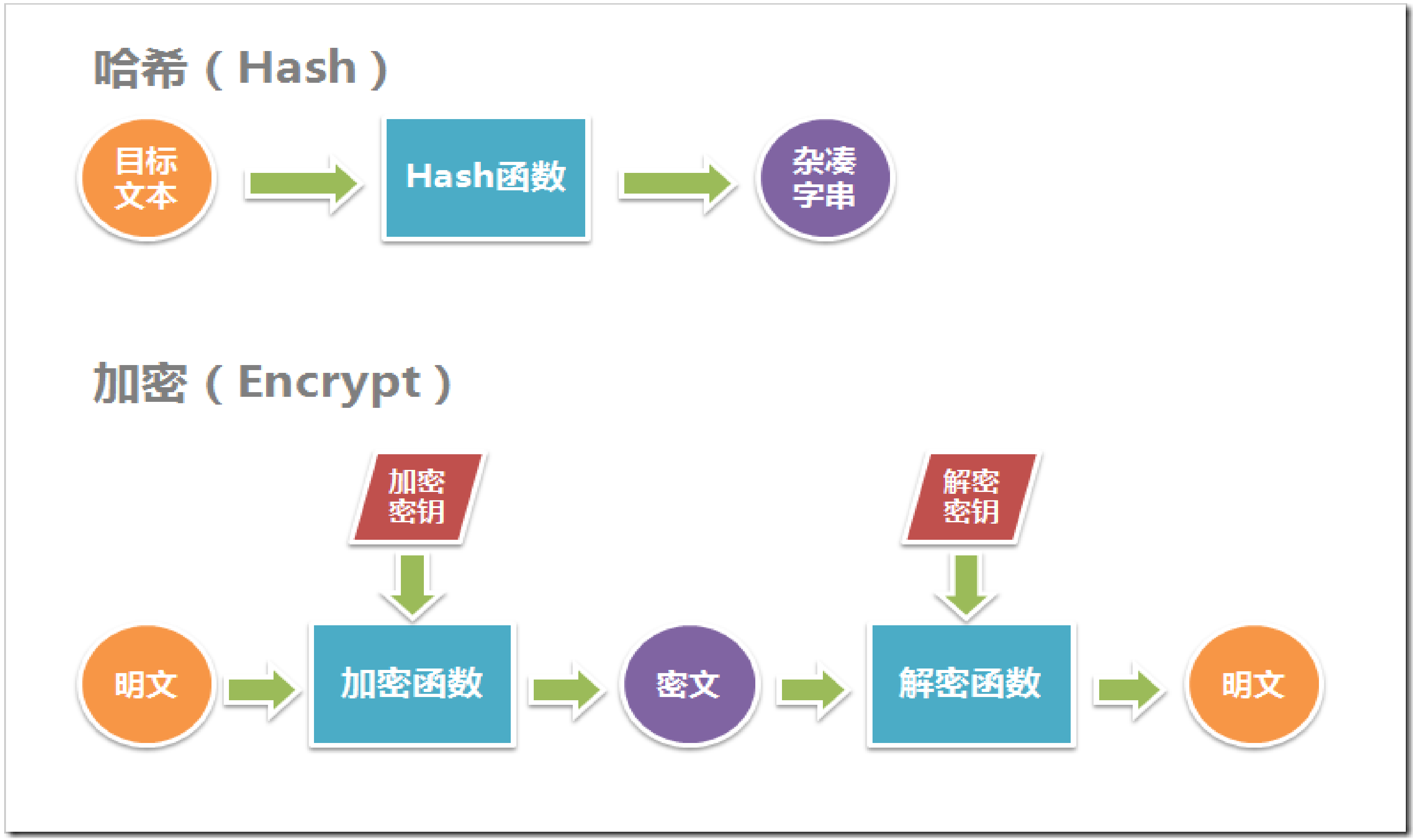
- **哈希与加密的数学基础**

  从数学角度，哈希和加密都是一个映射。两者的正式定义分别为：

  一个哈希算法 是一个多对一映射，给定目标文本S，H可以将其唯一映射为R，并且对于所有S，R具有相同的长度。由于是多对一映射，所以H不存在逆映射 使得R转换为唯一的S。

  一个加密算法 是一个一一映射，其中第二个参数叫做加密密钥，E可以将给定的明文S结合加密密钥Ke唯一映射为密文R，并且存在另一个一一映射 ，可以结合Kd将密文R唯一映射为对应明文S，其中Kd叫做解密密钥。

哈希(Hash)与加密(Encrypt)（4)



哈希(Hash)与加密(Encrypt)（5)

- **哈希（Hash）与加密（Encrypt）在软件开发中的应用**

　　哈希与加密在现代工程领域应用非常广泛，在计算机领域也发挥了很大作用，这里仅介绍在平常的软件开发中最常见的应用——数据保护。

　　所谓数据保护，是指在数据库被非法访问的情况下，保护敏感数据不被非法访问者直接获取。

　　要实现数据保护，可以选择使用哈希或加密两种方式。如何选择？

　　基本原则：如果被保护数据仅仅用作比较验证，以后不需要还原成明文，则使用哈希；如果被保护数据以后需要被还原成明文，则使用加密。

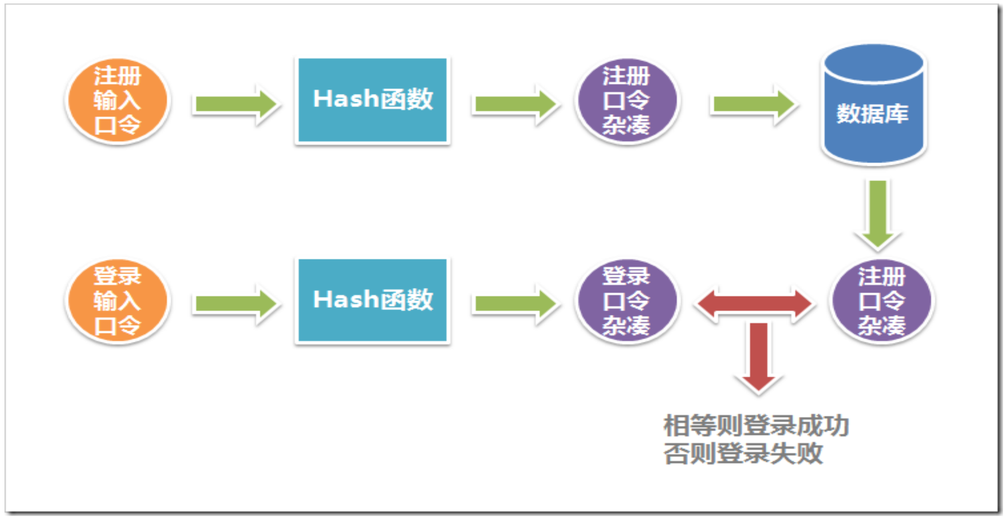
　　例如，开发一个系统，当用户忘记登录口令时，重置此用户口令为一个随机口令，而后将此随机口令发给用户，让其下次使用此口令登录，则适合使用哈希。实际上很多网站都是这么操作的，这是因为注册时输入的口令被哈希后存储在数据库里，而哈希算法不可逆，即使是网站管理员也不可能通过哈希结果复原你的口令，而只能重置口令。

　　相反，如果系统要求在用户忘记口令的时候必须将原口令发送给用户，而不是重置其口令，则必须选择加密而不是哈希。

哈希(Hash)与加密(Encrypt)（6)

- **哈希（Hash）与加密（Encrypt）在软件开发中的应用（续）**

1、很多人使用一次哈希进行数据保护的方法，其原理如下图所示：



当前最常用的哈希算法是MD5和SHA1，.NET平台上用C#语言实现了MD5和SHA1哈希的代码，由于.NET对于这两个哈希算法已经进行了很好的封装，因此用户不必自己实现其算法细节，直接调用相应的库函数即可（实际上MD5和SHA1算法都十分复杂，有兴趣的同学可以参考维基百科。