为了增加搜索entry的效率，我们才创造了一个由装载着entry的holders/BURCKETS组成的array

最终我们会创造一个table

如果对entry的插入被限制，使每个entry只能放置在这个数据结构中的一个可能位置(bucket)上，那么搜索entry就被限制为搜索特定的bucket，而不是搜索整个data structure

最好的方案是每一个bucket只存在了一个entry，这样搜索任意entry只用O（1）时间

为了达到这样的方案，我们需要吧key设置成独立整数，范围在[0，N-1]，n是array的size

这样会导致两个缺点：

如果N比存储的element实际数量大，造成不必要的浪费

key必须是整数，不一定能做到

对于后面一个缺点，如果我们们有一个限制size为n的array,必须将keys转换成0到n-1的整数

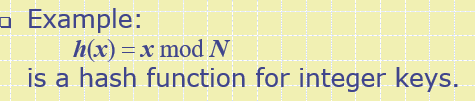
这个转换的method就叫hash function

整个数据结构（array和holders\buckets）叫做hash table

注意这个转换有可能吧不同的key转换成相同的index value，将多个entry放入同一个bucket这叫做collision//碰撞

Hash function and hash tables

hash function h 会把key转化到[0，N-1]

例如就是整数key 的hash function

最终求到的结果integer h（x）就是key x的hash value

一个限制key type的hash table由以下部分组成

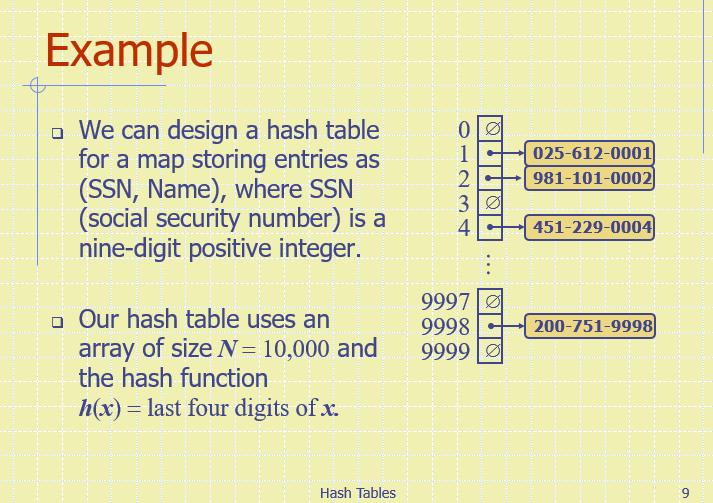
hash function h

size为N个holders/busckets of entries的Array(也叫table)

当通过hash table构建一个map的时候，我们的目的是把entry k,v()，存储在index I =h(k)上

换句话说，entry(k,v)被存储在A[(h（k）)]中，即entry被经过hashfunction存在对应的array index里

例如



我们设置一个hash table来存储entry(SSN,NAME)

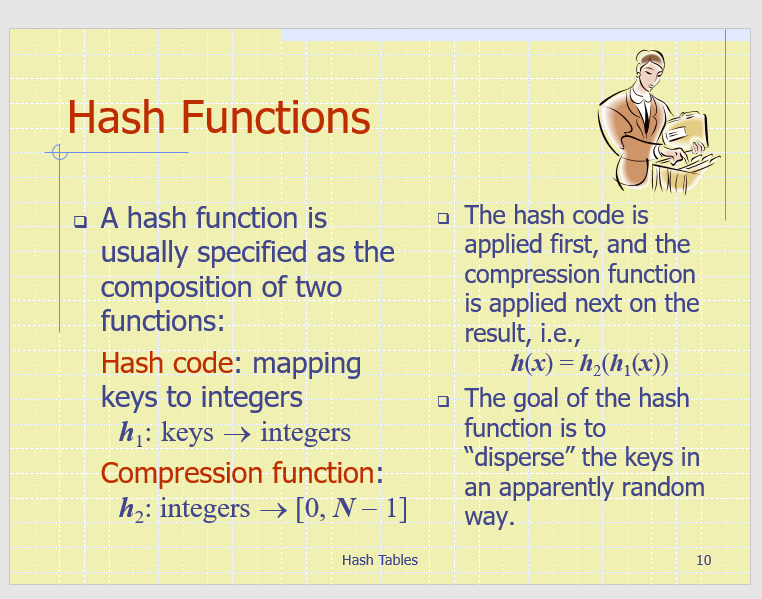
H（x）,最后四位digit

hash function

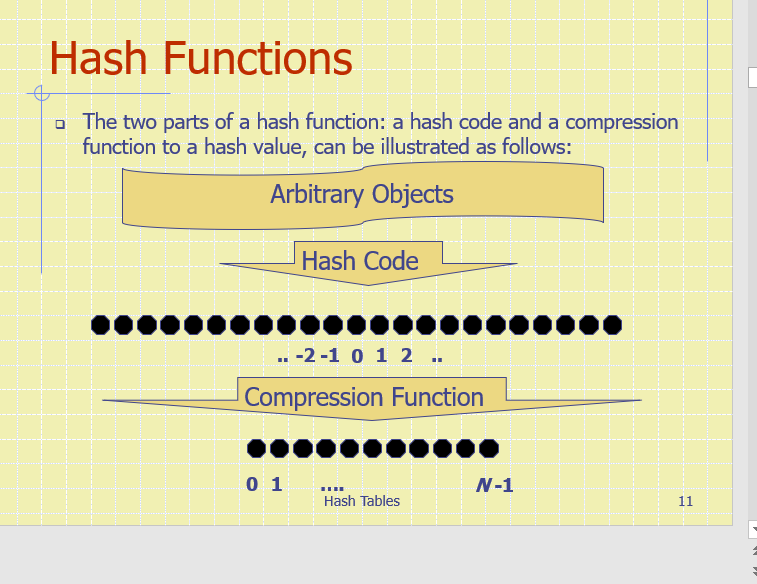
hash function由两个function组成

1. hash code:把keys转换成Integer
2. compression function:压缩公式，可以把Integer压缩到[0,N-1]

先用hash code,再用compression function



hash function的目的是以明显的随机的方式分散key



hash code 需要尽量避免collision/多个entry有同个Key

另外，同样的原始key应该对应成同样的hash code//而不是随机

转换hash code有多种方式：

1. 用memory address(java 里的hash code)
2. Integer Casting//转换成整数 用(Integer)这种
3. Summing component //求和组件
4. polynomial accumulation // 多项式积累

memory address法：

我们把key object的内存地址重新看作是一个整数

java用的就是这个hashcode法，object() class里有hashcode() method, 会return一个32位integer(内存地址)，是所有java object的默认hash code

一般来说不错，但是对数字与string key不行

两个同样的string key有不同的address，这样同样的Key就会产生不同的hash code，不能满足

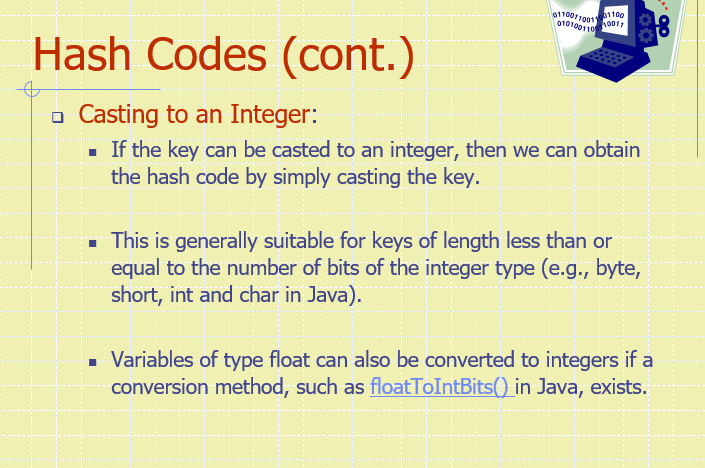
：

Casting to an integer法

如果key可以被cast成integer，那么我们就可以轻松得到一个hash code

这个方法适用于key--------长度小于或等于integer type的bit数量 （例如，byte，short,int and char）

浮点数也可以被转换成Integer如果存在 conversion method例如



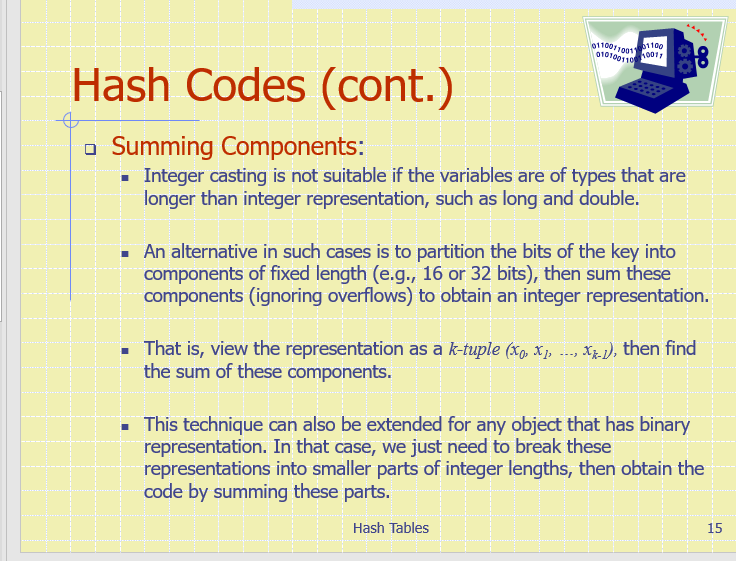
summing component:

Integer casting 对那些长度高于integer表达法的variable不适用，例如long与double

替代方法是将key的各个bit划分为多个固定长度的模块（例如16bit或32bit），然后把这些模块加到一起（忽略overflow）来得到一个整数表达

换句话说把原始表达切割成k份（x0,x1,…xk-1）然后找到这k份的和

可以备用用到任意使用二进制表达法的object，我们只需要把原始数据切割成固定长度的part再相加就行



polynomial accumulation

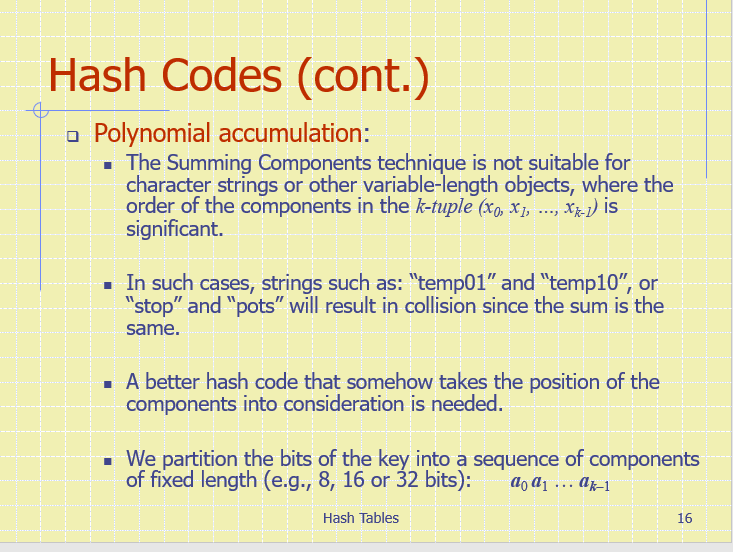
summing components技术不适用于字符串或其他可变长度object，其中k-tuple (x0、x1、…、xk-1)中的component

order非常重要。

这种情况下strings 比如“temp01”“temp10”或”stop”和“pots”和相同，会导致collision

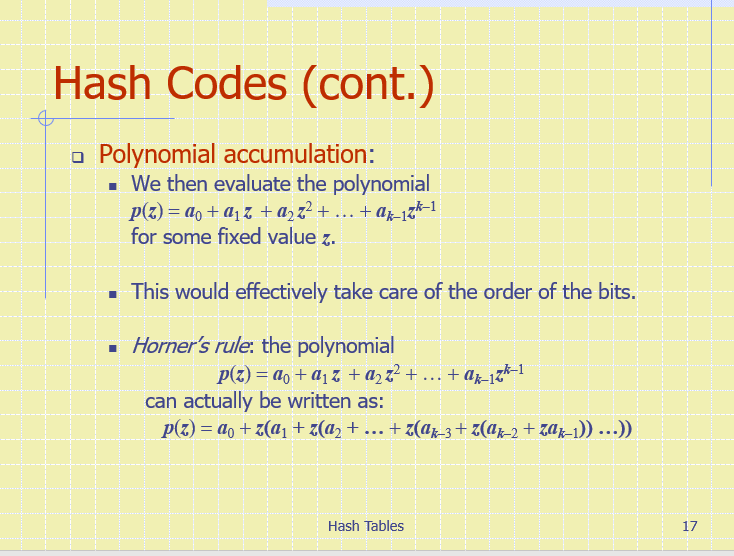
因为他们顺序不同但是每个字母大小相同，最后总和相同

作为替代，我们把key的bits们分为一系列的固定长度的component



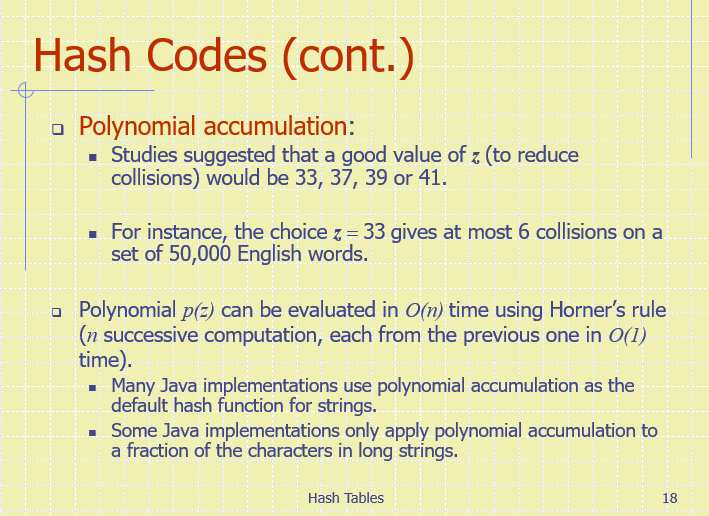
然后计算多项式，这样就考虑了顺序，因为每个component指数不同

horner rule:另一种写法



研究表明最好的z是33.37,39或41，他们能尽量减少collision

如果POLYNOMIALL P(z）用hornor表示，只需要时间是O（n）



Compression function

一个优秀的compression需要缩小有可能的collision数量

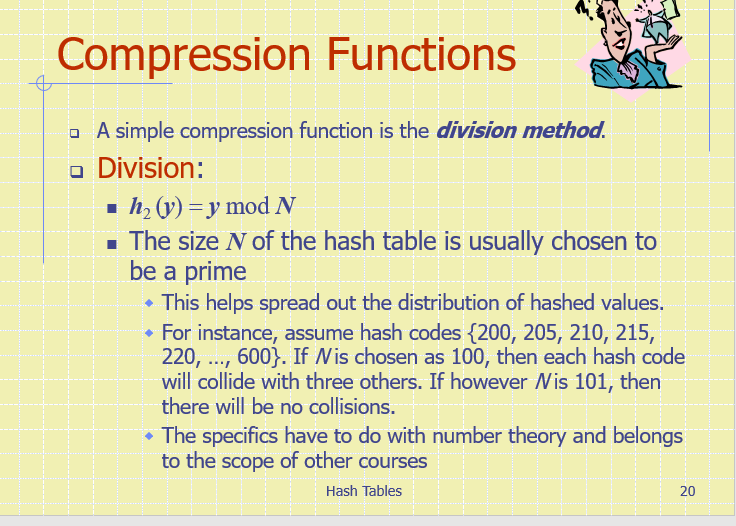
最基本的compression function叫做division method 、

N指的是hash table的size，通常是一个质数，

这有助于分散hashed codes，

例如加入经过第一层转换的hash code是{200,205,210,215,220。。600} 然后N设置成100，那么每一个hash code都会与另外三个collide例如，205,305,405,505.如果N=101，那么就没有collision

例外是200 300 400 500 600，有四个collide

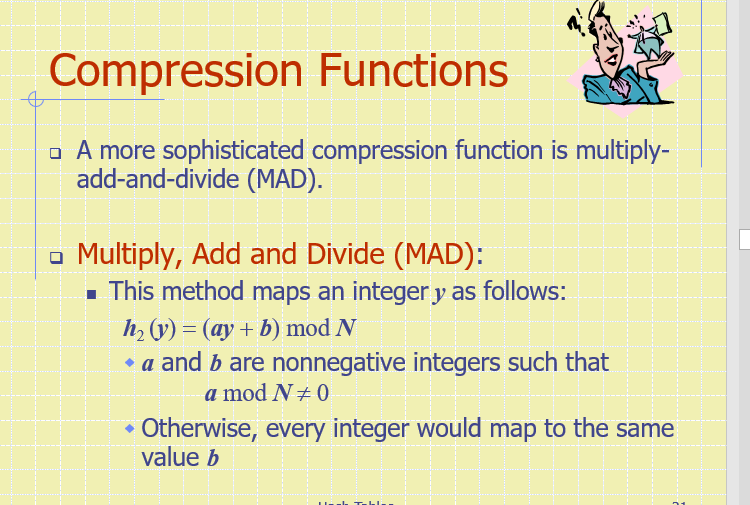


compression function

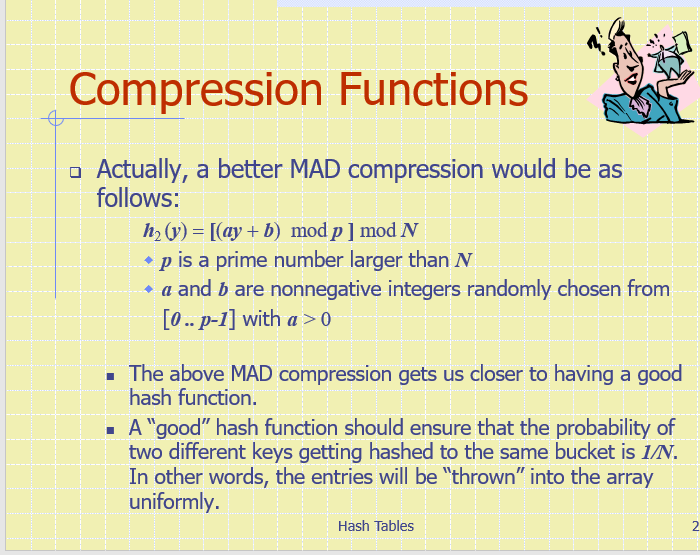
一个更复杂的方法是MAD

ay+b mod N,a与b是非负实数，a modN不等于0

否则，每个整数都会映射到相同的值b,因为ay mod N永远等于0



进阶的mad法



p是一个大于N的prime number

a 与b是0到P-1之间随意选取的非零整数，a>0

hashf function的 good：所有的entry被均匀的分配在不同的Index中，即0里面有4个entry，，那么1里也应该有4个entry

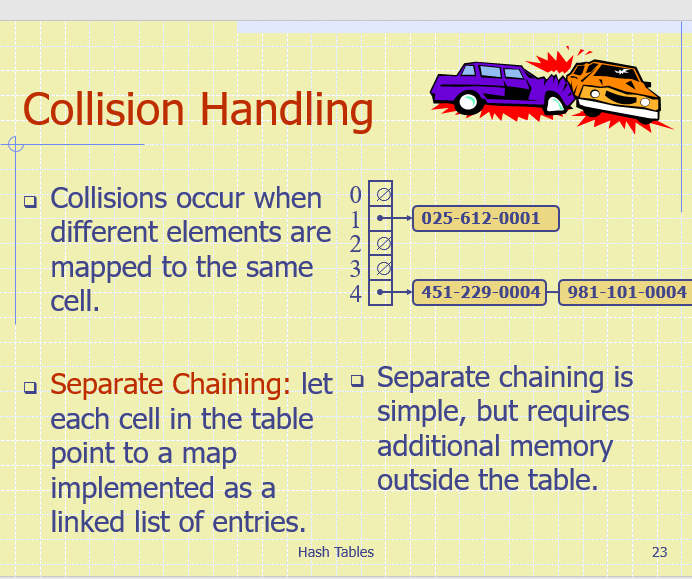
一个优秀的hash function能确保两个不同的key hash到同一个bucket里的几率是1/N，换句话说，entry们将被统一的扔到array里

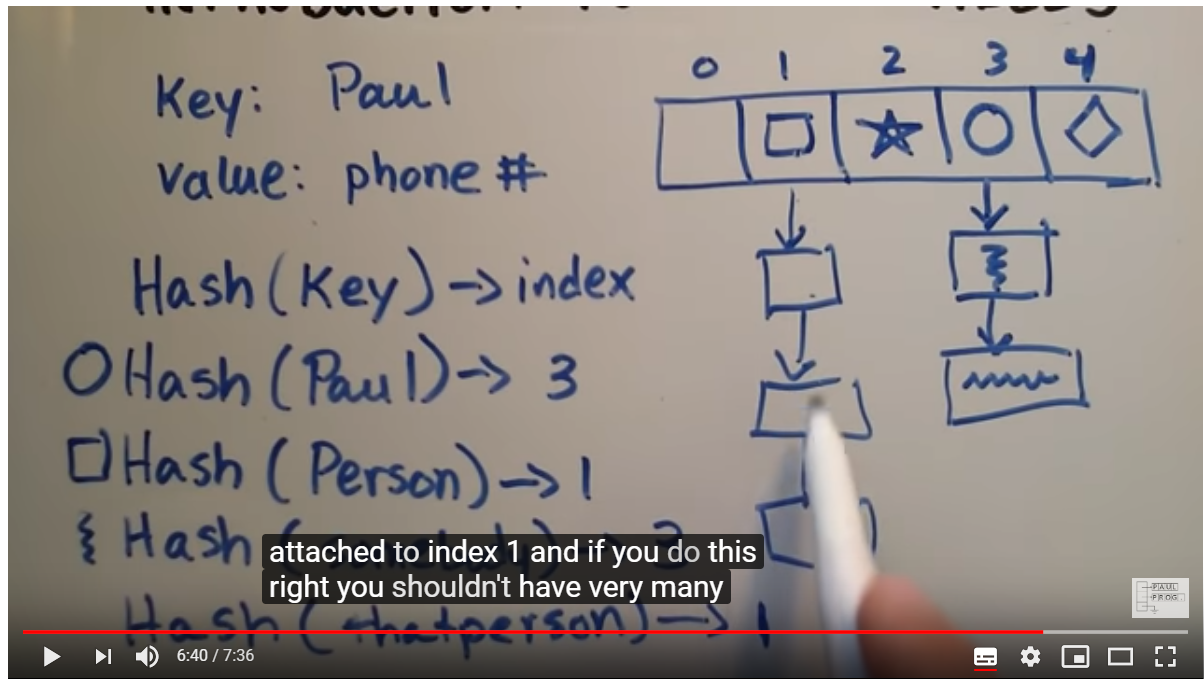
collision handling:控制collision

但不同的element被map到同一个格子里的时候就是collision

separate chaining：让每一个table里的cell指向由linked lisst构建的map

separate chaining简单，但是需要table以外的memory

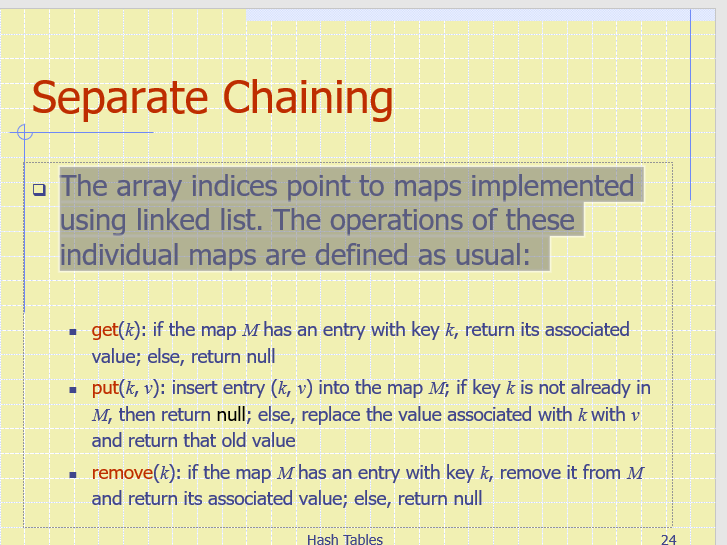


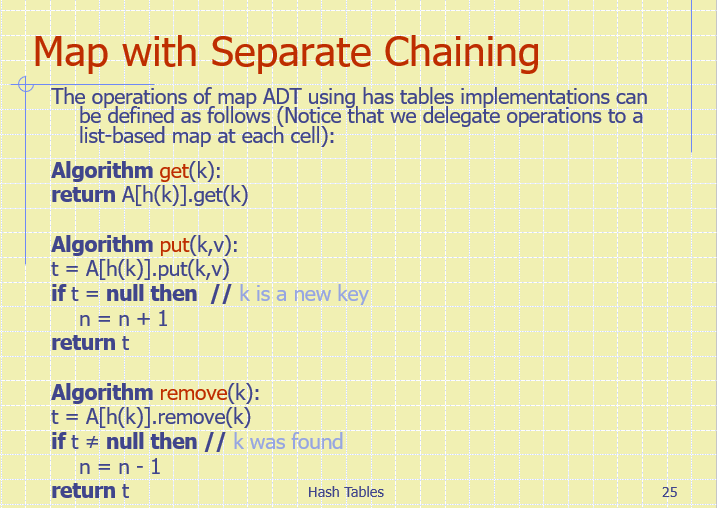


当一个index满了，再插入一个新元素，就用linked list把他连起来，组成一个map

hash map的思路就是通过hash公式可以直接找到他在array第几格，减少时间

array索引指向Linked list构成的Map。这些独立map可以进行的操作与普通map相同



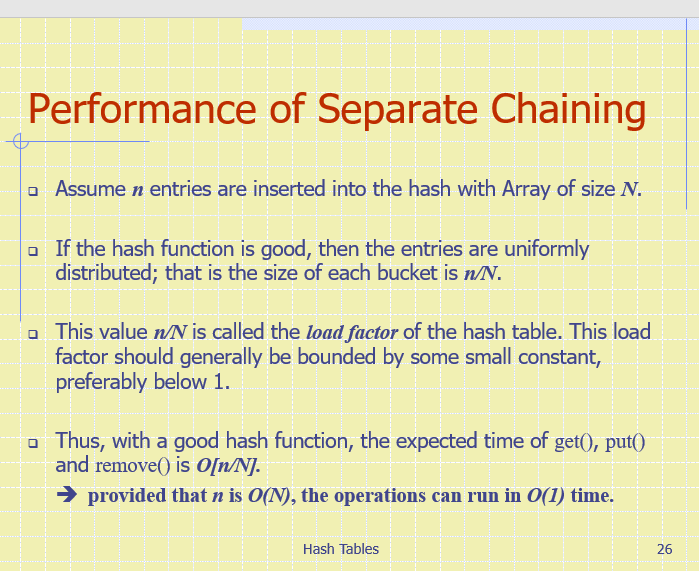


假设n个entry被insert到一个大小为N的array里，

如果hash function good，那么entry被均匀分配，每一个bucket size的大小为n/N

n/N的值叫做load factor： 我们应该尽量让load factor约束在某一个较小的常数下，最好小于1

英雌，一个优秀的hash table，get（）,put() remove都应该是，我们只用O（1）找到对应index，然后在n/N大小的linked list里进行操作



Open address:load factor小于1 即每个bucket只有一个entry的hash table

separate chaning有很多好处，但是有一个小小不足，

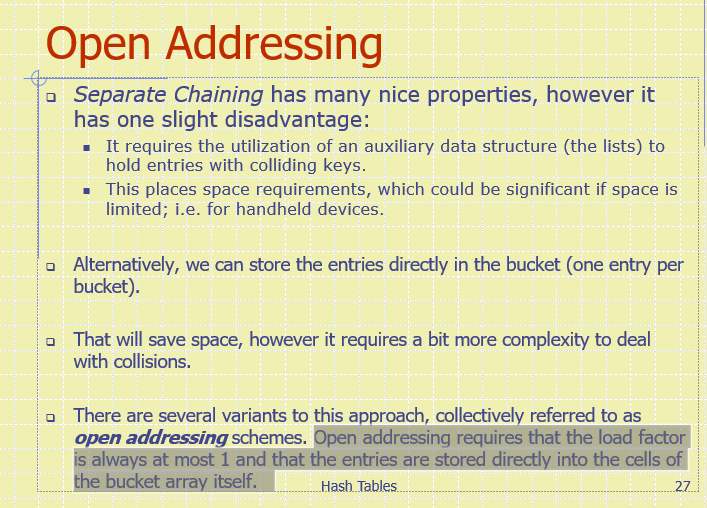
他需要一个辅助数据结构（LIST）来保存 colliding keys的entry们

额外数据结构将会需求巨大的space如果主程序的space是limited

作为替代，我们可以直接把entry存放在bucket里，一个·entry一个bucket

他会减少space使用，但是需要更多时间来解决collision

有好几种方案能达到open addressing，统称为open addressing scheme., open address 需要load factor最大也只能到1，entry被直接存入array中的bucket中



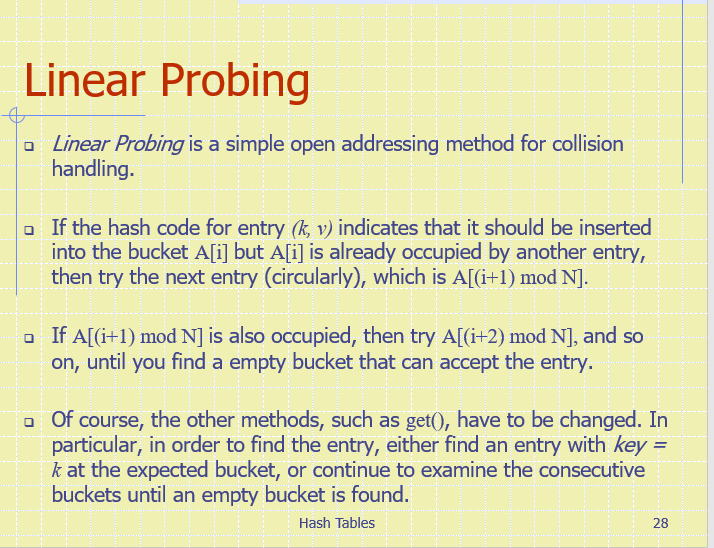
Linear probing 线性探查

linear probing是一种用来collision handling的open addressing method

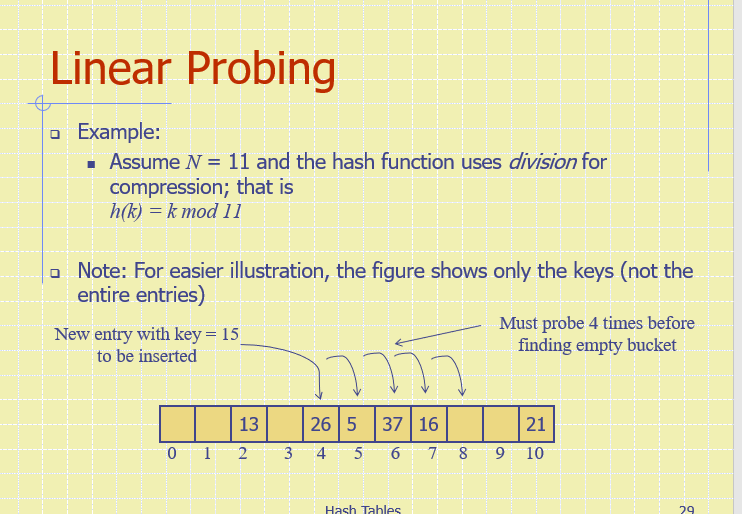
如果entry（k,v）的hash code告诉你这个entry应该被从插入bucketA[i]而A[I]已经有东西了，那么就尝试另一个格子A（i+1） mod N, //mod N是为了循环

如果A+1 MOD N也被占据了，那么就I+2,直到找到一个empty bucket。

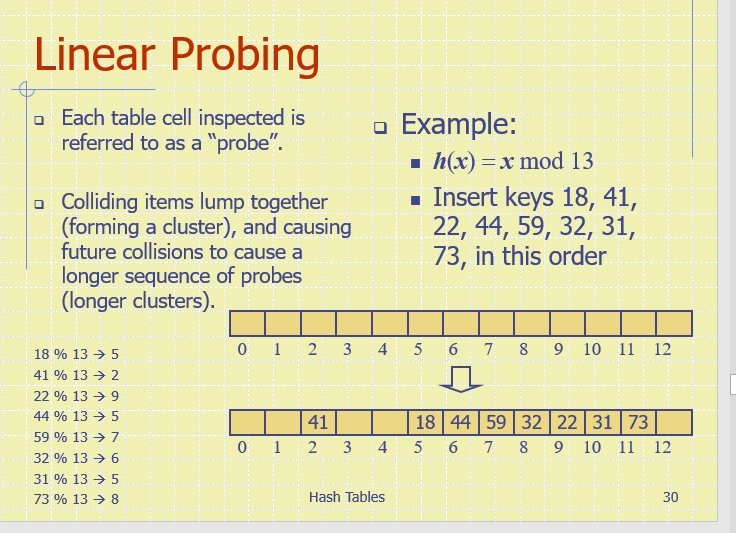
对应的，其他Method例如get也需要做出改变。为了找到对应entry，假设key=k，我们有可能找到所需entry，也有可能找不到我们要的entry(因为在插入的时候已被别的entry占据，所以往后放置了)还需要继续验证整个bucket直到遇到empty bucket



实例，hash function使用division，插入15,15mod11=4,我们应该插到4,但是4已经有26了，所以一直往后插入到空index 8

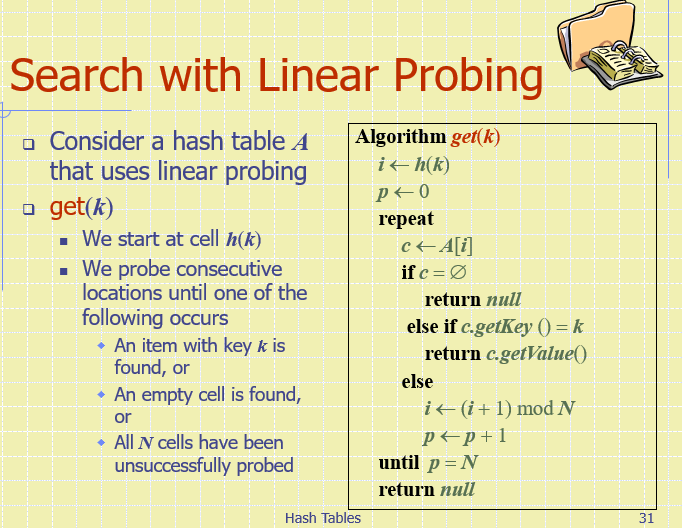


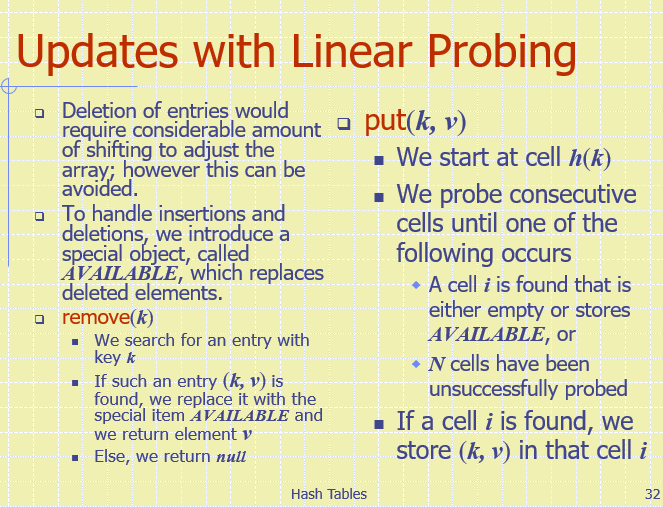
每一个被检查的table cell都被称作为probe探针



get 从cellh(k)开始

h(k)就是hash 后的第一个对应key，i代表初始index，p代表0，c是A[i]，如果c等于null，说明不存在，如果不是null，当前key=k，那么就是所求值，当前Key不等于K，那么就加1,做这个循环一直到p=N,这时说明整个都循环过一次了，还是没有return null

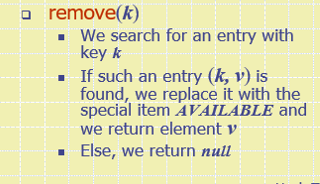


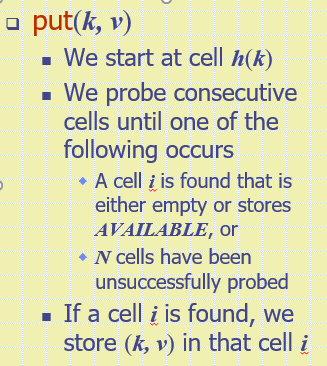


update with linear probing 线性探针更新

删除entry 需要大量的shifting（比如18Mod5,23mod5,28mod5,都应该在index3,当我们删除18,23应该占据18的位置，28应该占据23的位置），然而这是可以避免的

为了控制insertion 和deletion,我们引入一个特殊object，叫做AVAILABLE,用来代替被删除的element

找到key为k的entry，找到他的时候我们用特殊item available来代替他然后return element v

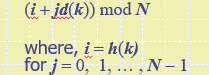
从h(k)开始·我们探测连续的单元格，直到出现以下情况之一：一个cell i是empty或者存着AVAILABLE

或N个cell都没有成功探测到， 如果cell i找到了，我们把(k，v)存储在cell i 里

double hashing,

double hashing有个次级hash function d(k)

double hashing来控制collision重叠的方法也是吧item放在第一个可用的cell



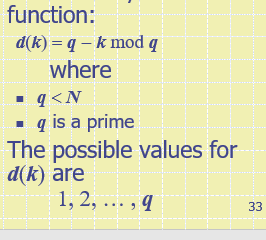
次级funcction d(k)不能等于0

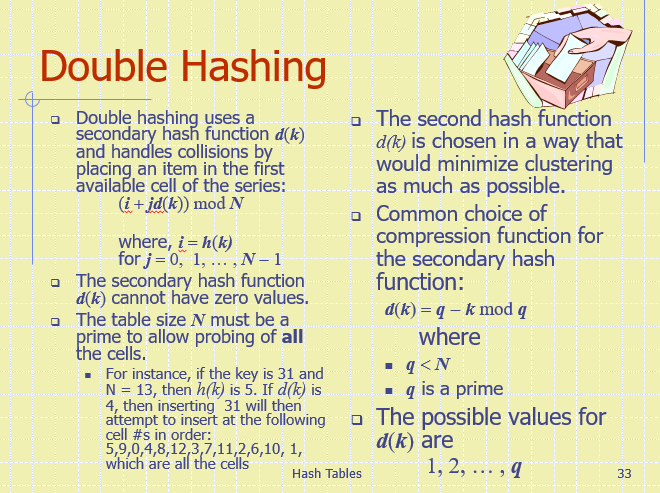
table的size N必须是一个质数这样才能probing所有的cell

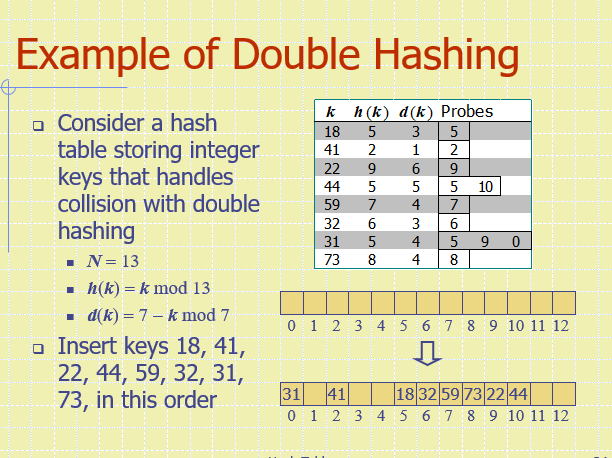
加入，Key是31，N=13，然后h(k)第一步得到的是5，如果d(k)是4，那么我们insert的顺序是

5,9,0,4,8,12,3,7,11,2,6,10,1，没有重复并且用了所有index

second hash function d(k)的选取方式:尽可能的把所有的entry归拢起来

标准的选secondary hash的方法





7是我们选的prime number， h(k)=key mod13 , d(k)=7-key mod 7 ,hash table长度为3

对应的probe h(k)+jd(k),

首先如果h(k)取得的地方是空，就第一步h(k)就完事儿了，只有当h(k)被占据的时候才加d(k

)

换句话来说,在原先的linear probing中，被占据的时候只能一个一个往前累加，而double hashin允许你以d(k) 跳着加

quadratic probing

另一种open addressing的方法

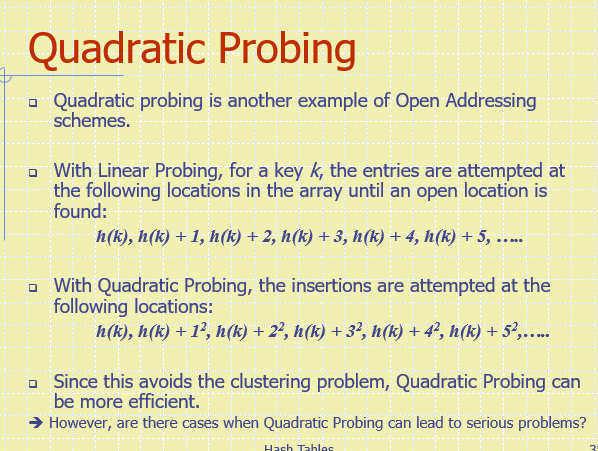
linear probing：对应一个key k， entry将会一个一个查找直到找到open location

h(k)+1\2\3\4\5\6

quadric probing可以加的是平方

H(k)+1^2\2^2\3^2

quadric probing规避了clustering problem,更有效率，但有可能产生了其他问题



open addressing分析

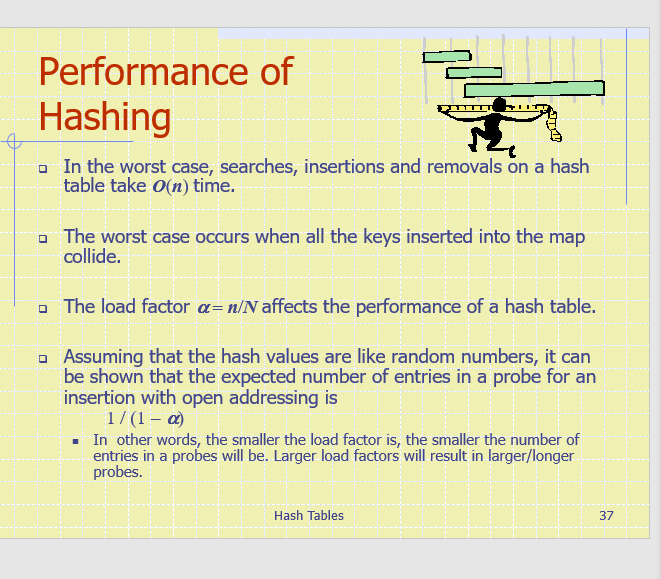
Open addressing和separate chaining比，save了空间（chaining需要用额外的List结构）

然而这并不意味着更快，实际上separate有时候比他还快

而且使用separate chaining，你可以插入任意数量的entry而不是在意array bucket的size

如果我们不在意空间使用，separate chaining是我们更常用的方法

hashing的performance

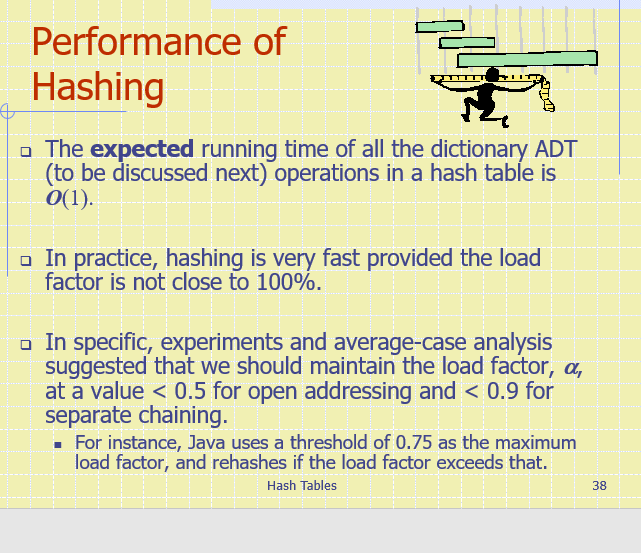


最差的就是O（n）。所有key都插在一个地方导致collision

，load factor影响了hash table的快慢

假设一个hash table是优秀的，（每个entry插入任意地方几率相等，并且每个地方拥有entry数量相同）那么他几率是1/1-a

换句话说，load factor越小，里面的entry越少



我们期待的running time是O（1）

实际上，如果load factor 不接近100％，hashing非常快

实际上opendressing，我们应该让load factor<0.5,separate chaining需<0.9

rehashing :如果load factor超过了界限，我们应该resize整个table,通常来说我们直接double，我们用一个新array的时候，应该重新设计hash function

