数据结构与算法

Data Structure and Algorithm

XII. 哈希表（上）

授课人：Kevin Feng

翻译 ：王 落 桐

万门大学 www.wanmen.org

课前回顾

- 数据结构及算法
- 数学回顾
- 数组（Array）和数组列表（Array List）
- 递归 vs. 迭代
- 二分法搜索
- 分治法
- 链表
- 栈和队列

万门大学 www.wanmen.org

CONTENTS

目录

- 哈希表
  - 数据成员（Data Member）
  - 操作（Operations）
  - 哈希（Hash）
  - 冲突（Collisions）
  - 解决方案（Resolution）

万门大学 www.wanmen.org

哈希表源起 | Motivation

- 数组查找：线性增加
- 如何改进？
- 我们可以通过数组索引，直接访问块（block），这种方法的访问时间是常数。
- 数组：
    - 牺牲空间换取时间——"holes（空穴）"会吃掉很多存储空间
    - 依赖于元素之间的顺序，元素之间的顺序将会转化为数据存储在内存空间上的顺序
- 还可以进一步改进么？
- 这个可以有，哈希大法好！

基础想法

- 关联数组（Associative Array），映射（Map），特征表（Symbol Table），字典（Dictionary）
- 字典是怎么工作的？
- 由（关键字，值）（key，value）对组成
- 理想情况下，key是没有重复的
- 操作：
  - 加入一对
  - 删除一对
  - 修改现存对当中的值
  - 查找特定关键字对应的值

数学相关

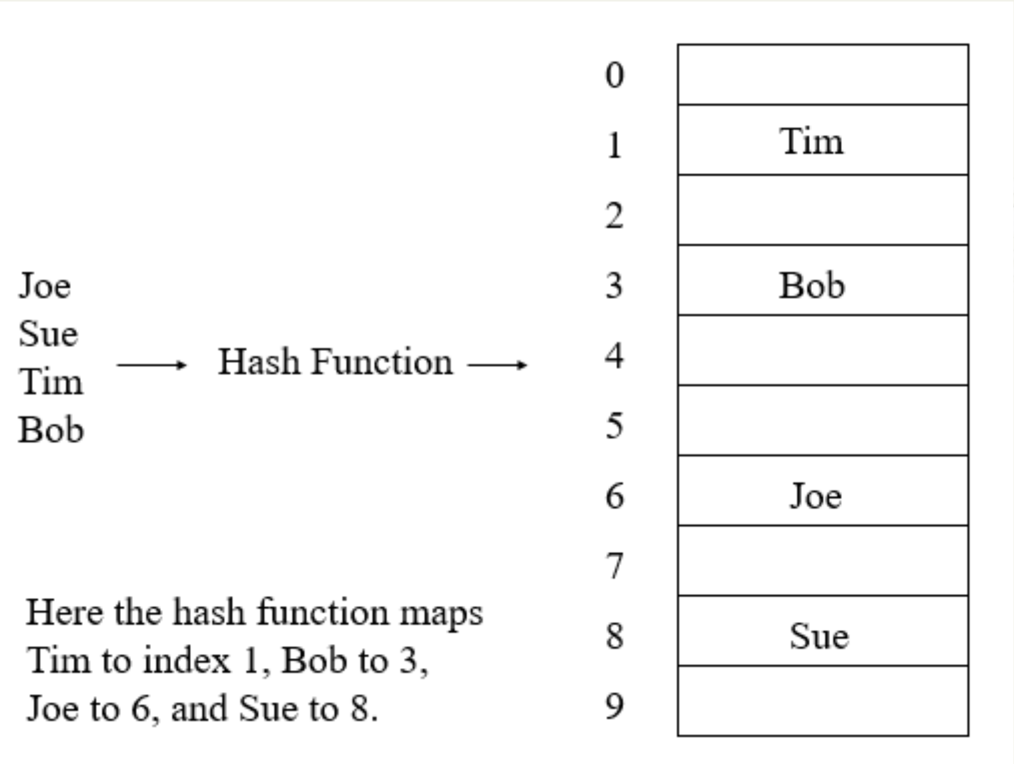
- 通用表示：
  - U——所有可能取到的值的结合
  - K——存储在字典中的所有关键字的结合
  - |K| = n
- 当U很大时：
  - Array是不实际的
  - |K| << |U|
- 使用一种大小与|K|成比例的表格——哈希表
  - ！这种形式会失去直接寻址的能力
  - 定义一个可以将关键字（keys）映射到哈希表的槽（slots）的函数

哈希基础

- 通用以下方法实现直接索引：
  - 划分一块足够的存储空间（block）给所有有可能出现的数据
  - 给所有的数据分配存储空间，存储地址由其原始地址经过简单的计算得来
- 哈希函数可以将任意关键字映射到一个有效的array位置
- 对于整数型关键字，（key mod N，mod：取模）是最简单的哈希函数
  - SSN（社保号码），电话号码，邮政编码
- 通常来说，只要能将关键字空间映射到数组索引（array indices）空间上的函数就是有效的
- 优秀的哈希函数会将数据均匀的映射到数组上
- 哈希函数 h：将U映射到哈希表的槽（slots） T[0…m-1]
  - $h : U \to \{0, 1, \cdots, m-1\}$
  - 对于array来说，关键字k映射到slot A[k]
  - 对于哈希表来说，关键字k映射/哈希到slot T[h[k]]
  - h[k] 是关键字k的哈希值

简化

- 现有一个盛有许多关键字（keys）的数组，假设是银行账户的账户名
- 每一个key可以映射到[0 , array_size-1]
- 这个映射函数就是哈希函数（Hash Function）
- 哈希函数应该尽可能的将keys均匀的分配到存储单元（cells）中
  - 右图：哈希函数将Tim映射到了索引1，Bob到了索引3，Joe到了索引6，Sue到了索引8



计算哈希函数

- 理想状态下：重洗keys然后均匀赋给这些关键字以表格索引（table index）
  - 高效的计算
  - 每一个table index同等概率分配给每一个key
  - 全面搜索问题在实际的应用中仍然存在诸多问题
- 第一个例子：电话号码
- 第二个例子：社保号码
  - 差：前三位数字
  - 好一些：末尾三位数字
- 第三个例子：人
  - 姓，名
  - 生日
- 实际的挑战：对于不同的类型的key需要不一样的方法。

Python 哈希惯例

- 一个哈希类必须满足在它的一生中有一个不变的哈希值（需要一个__hash__()的方法），并且可以与其他类进行比较（需要一个__eq__()的方法）。相同的哈希类（Hashable objects）必须有相同的哈希值
- 要求：
  - 如果x=y，那么x和y有相同的哈希码（hash code）
  - 理想状态，如果x ！= y，x和y的hash code不同。
- 默认设置：x的内存地址
- 合法（但是表现差劲）的设置：总是返回17.
- 个性化设置：整型（Integer），浮点（Double），字符串（String），文件（File），超级链接（URL），日期（Date）……
- 用户定义的类型：靠自己
- Python内任何不可变的内置类（immutable built-in objects）都是哈希类，可变的容器（列表，字典）都不是。默认的用户定义的实例是哈希类，这些实例之间是不相等的（除了自己和自己）并且这些实例的哈希值由他们的id（）产生

Hash Code Design

- "Standard" recipe for user-defined types
  - Combine each significant field using the 31x + y rule (Horner Rule)
  - If field is a primitive type, use wrapper type hashCode()
  - If field is null, return 0
  - If field is a reference type, use hashCode() ← applies rule recursively
  - If field is an array, apply to each entry ← or use Arrays.deepHashCode()

- Horner's Rule gives us a simple way to compute a polynomial using multiplication and addition:

$$a_0 + a_1x + a_2x^2 \cdots + a_nx^n = a_0 + x(a_1 + x(a_2 + \cdots + xa_n)\cdots)$$

- In practice. Recipe works reasonably well; used in Java libraries
- Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Collision

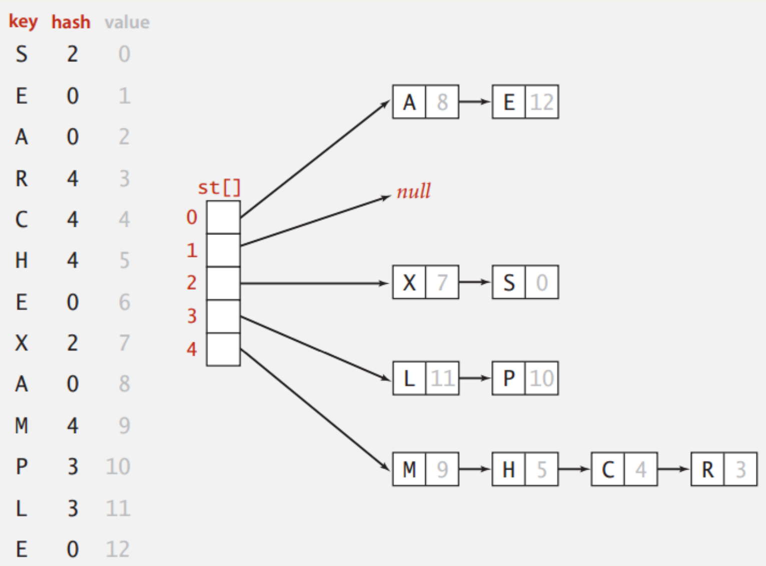
- What if we use a block which is not big enough for all possible items?
  - Addressing function must map all items into this space.
  - Some items may get mapped to the same position ⇒ called a *collision*.
- A collision occurs when h(x) maps two keys to the same location.
- Challenge: Deal with collisions efficiently
- Collision Resolution
  - Separate Chaining
  - Probing, Open Addressing

Bucketing and Separate Chaining

- Allow more than one item to be stored at each position in the hash table
  - → associate a List with each hash table cell. . .
- Bucketing
  - Each list is represented by a fixed size block
- Advantages
  - Simple to implement: hash to address, then search list
- Disadvantages
  - Searching the list slows down Table access
  - Fixed size → may waste a lot of space
  - Buckets may overflow → back where we started, a collision is just an overflow with a bucket size of 1

Separate Chaining

- Allow more than one item to be stored at each position in the hash table
  - ➔ each list is represented by linked list or chain
- Use an array of M < N linked lists
  - Hash: map key to integer i between 0 and M - 1
  - Insert: put at front of i th chain (if not already there).
  - Search: need to search only i th chain.
- Advantages
  - Simple to implement: hash to address, then search list
  - No overflow
- Disadvantages
  - Searching the list slows down Table access
  - Extra space for pointers (if we are storing records of information the space used by pointers will generally be small compared to the total space used)
  - Performance deteriorates as chain lengths increase

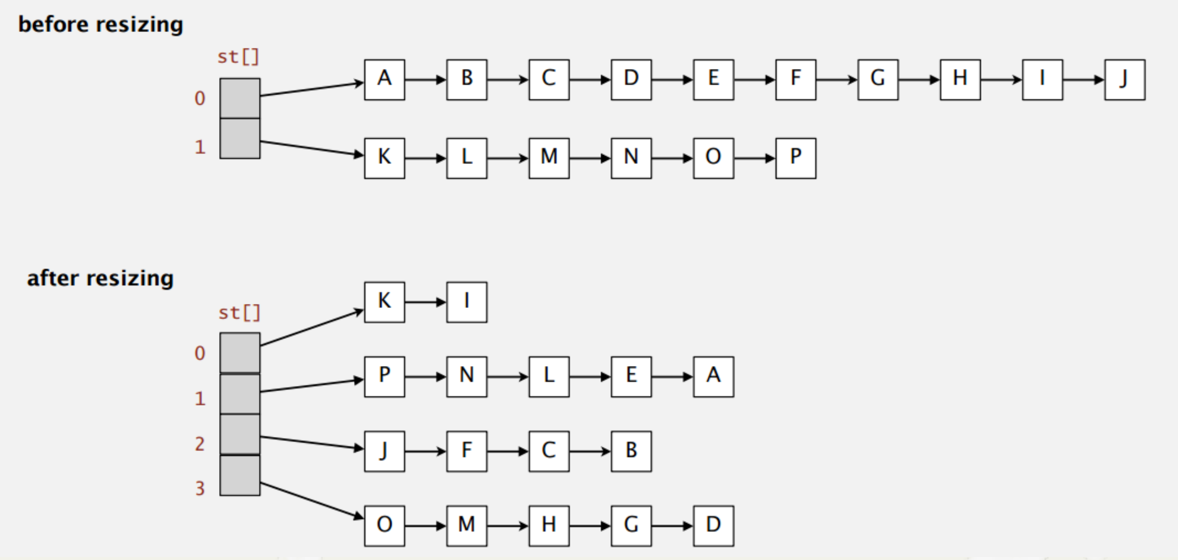


Performance

- Worst Case
  - all items stored in a single chain
  - $O(n)$ → same as List, no gain
- But expected case performance is much better. . .
- Load factor λ: the number of items N in the table divided by the size M of the table.
- Consequence: Number of probes for search/insert is proportional to N / M
  - M too large ⇒ too many empty chains
  - M too small ⇒ chains too long
  - Java: 0.75

Rehashing

- Worst Case
  - all items stored in a single chain
  - $O(n)$ → same as List, no gain
- But expected case performance is much better. . .
- Load factor λ: the number of items N in the table divided by the size M of the table.
- Goal. Average length of list N / M = constant
  - Double size of array M when $N / M \geq 8$
  - Halve size of array M when $N / M \leq 2$
  - Need to rehash all keys when resizing
  - *x.hashCode()* does not change
  - But *hash(x)* can change



Open Addressing

- When a new key collides, find next empty slot, and put it there
- If $h(x)$ is occupied, try $h(x)+f(i)$ *mod* N for $i = 1$ until an empty slot is found
- Many ways to choose a good f(i)
- Simplest method: Linear Probing
  - $f(i) = i$
- Linear-probing Hash Table
  - Hash. Map key to integer i between 0 and $M-1$.
  - Insert. Put at table index i if free; if not try $i+1$, $i+2$, etc.

Linear Probing

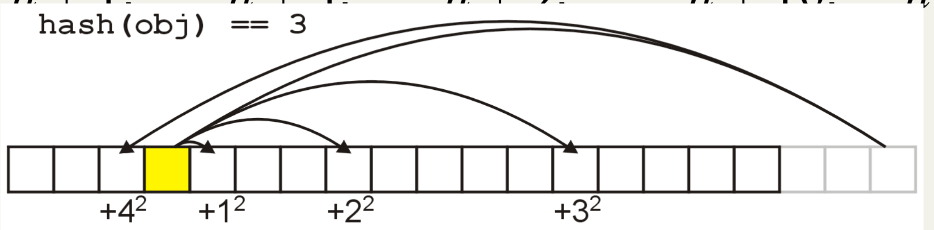
- Increment hash index by one (with wrap-around) until the item, or null, is found
- Removals
  - How do we delete when probing?
  - Lazy deletion: mark as deleted
  - We can overwrite it if inserting
  - But we know to keep looking if searching
- Primary Clustering
  - If there are many collisions, blocks of occupied cells form: primary clustering
  - Any hash value inside the cluster adds to the end of that cluster
  - it becomes more likely that the next hash value will collide with the cluster
- Instead of searching forward in a linear fashion, try to jump far enough out of the current (unknown) cluster

Quadratic Probing

- Increment hash index by one (with wrap-around) until the item, or null, is found
- Removals
  - How do we delete when probing?
  - Lazy deletion: mark as deleted
  - We can overwrite it if inserting
  - But we know to keep looking if searching
- Suppose that an element should appear in position h:
  - if h is occupied, then check the following sequence of the array:

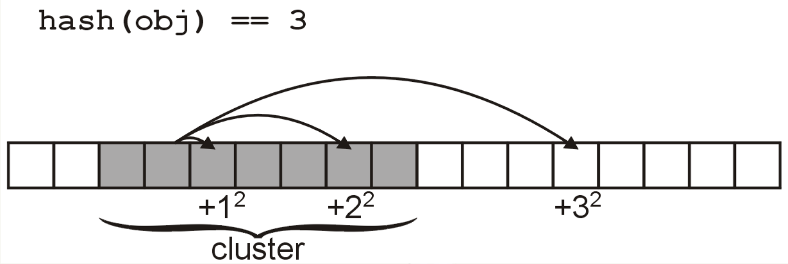
$$h+1^2,\quad h+2^2,\quad h+3^2,\quad h+4^2,\quad h+5^2,\ \ldots$$

$$h+1,\quad h+4,\quad h+9,\quad h+16,\quad h+25,\ \ldots$$



Quadratic Probing

- If one of $h + i^2$ falls into a cluster, this does not imply the next one will



- Even if two bins are initially close, the sequence in which subsequent bins are checked varies greatly
- Thus, quadratic probing solves the problem of primary clustering
- Unfortunately, there is a second problem which must be dealt with
- Suppose we have M = 8 cells
  - $1^2 \equiv 1, \; 2^2 \equiv 4, \; 3^2 \equiv 1$
- In this case, we are checking cell $h + 1$ twice having checked only one other cell

Quadratic Probing

- Unfortunately, there is no guarantee that

  $h + i^2 \bmod M$

  will cycle through 0, 1, ..., M – 1
- Solution:
  - require that M be prime
  - in this case, $h + i^2 \bmod M$ for $i = 0, \ldots, (M - 1)/2$ will cycle through exactly $(M + 1)/2$ values before repeating
- Example with M = 11:

  0, 1, 4, 9, 16 ≡ 5, 25 ≡ 3, 36 ≡ 3
- With M = 13:

  0, 1, 4, 9, 16 ≡ 3, 25 ≡ 12, 36 ≡ 10, 49 ≡ 10
- With M = 17:

  0, 1, 4, 9, 16, 25 ≡ 8, 36 ≡ 2, 49 ≡ 15, 64 ≡ 13, 81 ≡ 13

Quadratic Probing

- Thus, quadratic probing avoids primary clustering
- Unfortunately, we are not guaranteed that we will use all the cells
- In reality, if the hash function is reasonable, this is not a significant problem until λ approaches 1
- Secondary Clustering
  - The phenomenon of primary clustering will not occur with quadratic probing
  - However, if multiple items all hash to the same initial cell, the same sequence of numbers will be followed
  - This is termed secondary clustering
  - The effect is less significant than that of primary clustering

Double Hashing

- Double hashing uses a secondary hash function d(k) and handles collisions by placing an item in the first available cell of the series

  $(i + jd(k)) \bmod N$

  for $j = 0, 1, \cdots, N-1$

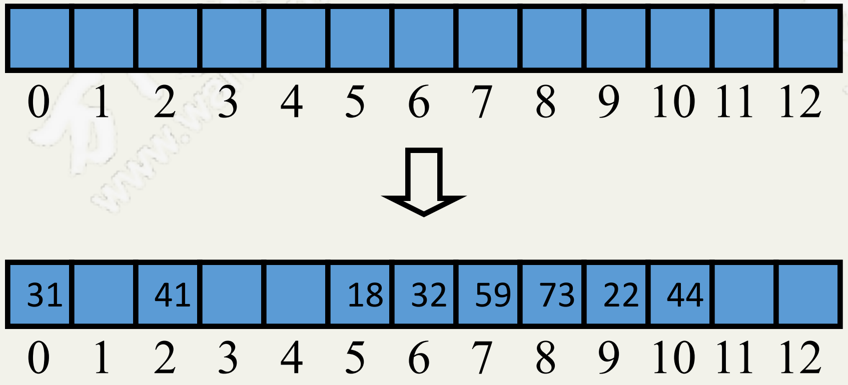
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression function for the secondary hash function:
  - $d_2(k) = q - (k \bmod q)$
  - where
    - $q < N$
    - q is a prime

www.wanmen.org

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Summary

- Hash tables can be used to:
  - improve the space requirements of some ADTs for which bounded representations are suitable
  - improve the time efficiency of some ADTs, such as Table, which require unbounded representations
  - In the worst case, searches, insertions and removals on a hash table take O(n) time
  - The worst case occurs when all the keys inserted into the map collide
  - The expected running time of all the dictionary ADT operations in a hash table is O(1)
  - In practice, hashing is very fast provided the load factor is not close to 100%
  - When the load gets too high, we can rehash….
- We have seen a number of methods for collision resolution in hash tables:
  - bucketing and separate chaining
  - open addressing, including linear probing and quadratic probing
  - double hashing

特别提醒

不要先哈希，后问问题

永远先问问题，然后哈希

哈希可以有加法！！！

对于树状结构，哈希会破坏其结构

万门大学 www.wanmen.org

实战！！

Are You Ready?

万门大学 www.wanmen.org

练习1

- 统计字母数
- 统计单词数
- 一个字符串中的首个独特的字
  - 给定一个字符串，找到第一个在这个字符串中没有重复的字并且返回其索引。如果不存在，返回-1
- 两个Array求交集
  - 给定两个Array，试写出函数来计算他们的交集
  - 结果中的元素必须是独一无二的
- 两个Array求交集（升级版）
  - 给定两个Array，试写出函数来计算他们的交集
  - 结果中的元素出现的次数和在两个array中出现的次数一样多

练习2

- 宝石和石头：
  - 字符串J代表是宝石的类型，字符串S是你拥有的石头。S中的每一个字符代表你拥有的一类石头，你想知道你手中的石头中有多少是宝石。
  - J中的字符是独特的并且在J和S中的是字母。字母是大小写敏感的，也就是说a和A是两个不同的类型。
- 包含冗余（217E）：
  - 给定的整型Array中如果出现了重复的元素则返回true，如果每一个元素都是独特的则返回false
- 包含冗余（219E）：
  - 给定的整型Array和整数k：试找出是否存在两个独特的索引（indices）i，使得num[i] = num[j]且i和j之间的距离小于k

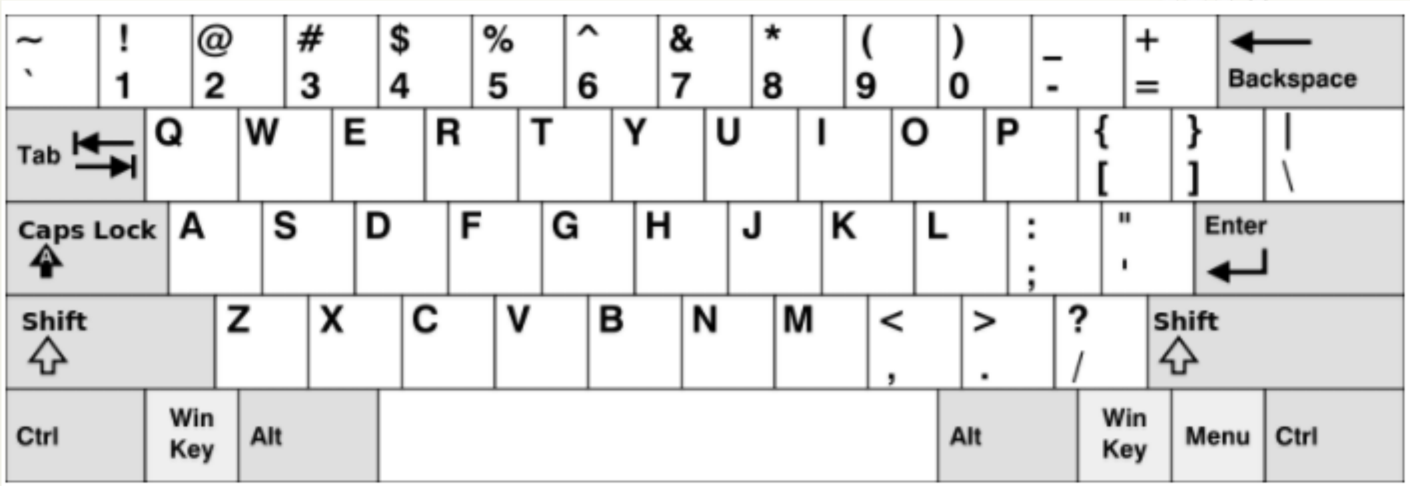
练习3

- 子域（subdomain）访问计数
  - 一个网站的域名形如“scholar.google.com” 由多种子域构成。在最上面的层级是“com”，在下一个层级是“google.com”，最底层的是“scholar.google.com”。当我们访问“scholar.google.com”的时候我们势必会f访问其父域 “google.com”和“com”
  - 现在定义“count-paired domain”（计数配对域）：次数+空格+地址。Eg：“9001 scholar.google.com”
  - 实例 1：
    - Input: ["9001 scholar.google.com"]
    - Output: ["9001 scholar.google.com", "9001 google.com", "9001 com"]
  - 实例 2：
    - Input: ["900 google.mail.com", "50 yahoo.com", "1 intel.mail.com", "5 wiki.org"]
    - Output: ["901 mail.com","50 yahoo.com","900 google.mail.com","5 wiki.org","5 org","1 intel.mail.com","951 com"]

练习4

◉ 键盘行

◉ 给定一个单词的List，返回可以用一行键盘字母输出的单词。键盘如下：



练习5

- 单词模式
  - 给定一种模式和一个字符串 str，判断str是否符合相应的规则
  - 例如：模式为在单词中间存在双射（bijection）并且在str中没有空单词
  - 实例：
    - pattern = "abba", str = "dog cat cat dog" should return true.
    - pattern = "abba", str = "dog cat cat fish" should return false.
    - pattern = "aaaa", str = "dog cat cat dog" should return false.
    - pattern = "abba", str = "dog dog dog dog" should return false.

练习6

- 两个List的最小Index（索引）
  - 假设Andy和Doris各有一个喜欢餐厅的清单，存储格式为string。
  - 请你帮她们找到她们都喜欢的餐厅：要求两个餐厅的Index相加最小，假设存在多对答案请将这些答案都输出出来，排序不分先后。
- 字典中的最长单词
  - 给出一个字符串数组words组成的一本英语词典。从中找出最长的一个单词，该单词是由words词典中其他单词逐步添加一个字母组成。若其中有多个可行的答案，则返回答案中字典序最小的单词。
  - 若没有答案返回空字符串

练习7

- 快乐数字
  - 写出判断数字是否是快乐数字的算法
  - 快乐数（happy number）有以下的特性：在给定的进位制下，该数字所有数位(digits)的平方和，得到的新数再次求所有数位的平方和，如此重复进行，最终结果必为1
  - 实例：19是一个快乐数字

$$1^2 + 9^2 = 82$$
$$8^2 + 2^2 = 68$$
$$6^2 + 8^2 = 100$$
$$1^2 + 0^2 + 0^2 = 1$$

练习8

- 有效字谜
  - 两个strings s, t, 写出一个可以判断t是否是s的有效字的算法
  - 例如:
    - s = "anagram", t = "nagaram", return true.
    - s = "rat", t = "car", return false.

- 在一个string中找到所有的有效字谜
  - String s 和 非空string p, 找到所有 p是s有效字谜的indices
  - s和p均有小写英文字母构成并且s和p都不长于20100
  - 实例 1:
    - Input: s: "cbaebabacd" p: "abc"
    - Output: [0, 6]
  - 实例 2:
    - Input: s: "abab" p: "ab"
    - Output: [0, 1, 2]

练习9

- 字谜组
  - 两个strings s, t, 写出一个可以判断t是否是s的有效字谜的算法
  - 给定一个array的strings, 将有效字谜组成组
  - Eg:
    - given: ["eat", "tea", "tan", "ate", "nat", "bat"], Return:
    - [
        ["ate", "eat","tea"],
        ["nat","tan"],
        ["bat"]
      ]

练习10

- 按照词频对字符进行排序
  - 给定string，按照词频降序排列
  - Eg 1：
    - Input: "tree"
    - Output: "eert"
  - Eg 2：
    - Input: "Aabb"
    - Output: "bbAa"

练习11

- 森林里的兔子
  - 森林里有若干兔子，每一只兔子身上都有某一种颜色。一些兔子（可能是所有的兔子）会告诉你森林中有多少和他们颜色一样的兔子。这些答案会被存储在array中。
  - 返回森林中可能有的最少的兔子的数量。
  - 例如：
    - Input: answers = [1, 1, 2]
    - Output: 5
  - 解释：
    - 两只答案是1的兔子可能是同一个颜色的，假设为红色
    - 答案是2的兔子不可能是红色的，假设这只兔子是蓝色的
    - 所以还有2只蓝色的兔子
    - 所以最少可能有5只兔子

练习12

- 砖墙
  - 给定一堵墙 wall，wall 分若干行，每一行等高，但每行可能由不同数量，不同宽度的 brick(s) 组成 ；求出自墙顶向下的一条垂直的路径，使路径经过的 brick 尽可能少，其中，若路径经过两块 brick 之间（故墙的左右两条边不算），则视为 其穿透而过，即此时路径经过的 brick 数为 0

```
Input:
[[1,2,2,1],
 [3,1,2],
 [1,3,2],
 [2,4],
 [3,1,2],
 [1,3,1,1]]
Output: 2
Explanation:
```

练习13

- 编码和解码 TinyURL
  - TinyURL是一个URL缩短服务，您可以在其中输入URL，https://leetcode.com/problems/design-tinyurl并返回一个简短的URL http://tinyurl.com/4e9iAk。
  - 设计TinyURL服务的方法encode和decode方法。编码/解码算法应该如何工作没有限制。您只需确保将URL编码为一个小型URL，并将该小型URL解码为原始URL。
- 公牛母牛
  - 程序随机生成4个 0到 9之间的整数，作为神秘数字。玩家通过反复的猜测找到这4个数。并且要求先后顺序也要正确，数值和位置都正确是公牛，数值正确，位置不对是母牛。如果相同的母牛重复出现多次只能算一个母牛，不能重复计次。
  - Eg：
  - 神秘数字: "1807"
  - 玩家猜测: "7810"
  - 输出: 1 bull and 3 cows. (The bull is 8, the cows are 0, 1 and 7.)

练习14

- 拥有最多点的直线（149H）
  - 二维平面中给出n个点，找到可以连起来最多点的直线

Summary

总结

- 哈希表
  - 数据成员（Data Member）
  - 操作（Operations）
  - 哈希（Hash）
  - 冲突（Collisions）
  - 解决方案（Resolution）
- 下节课
  - 树

万门大学 www.wanmen.org

数据结构与算法

Data Structure and Algorithm

XII. 哈希表

结束

授课人：Kevin Feng