数据结构与算法

DATA STRUCTURE

第十八讲 Hashing

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

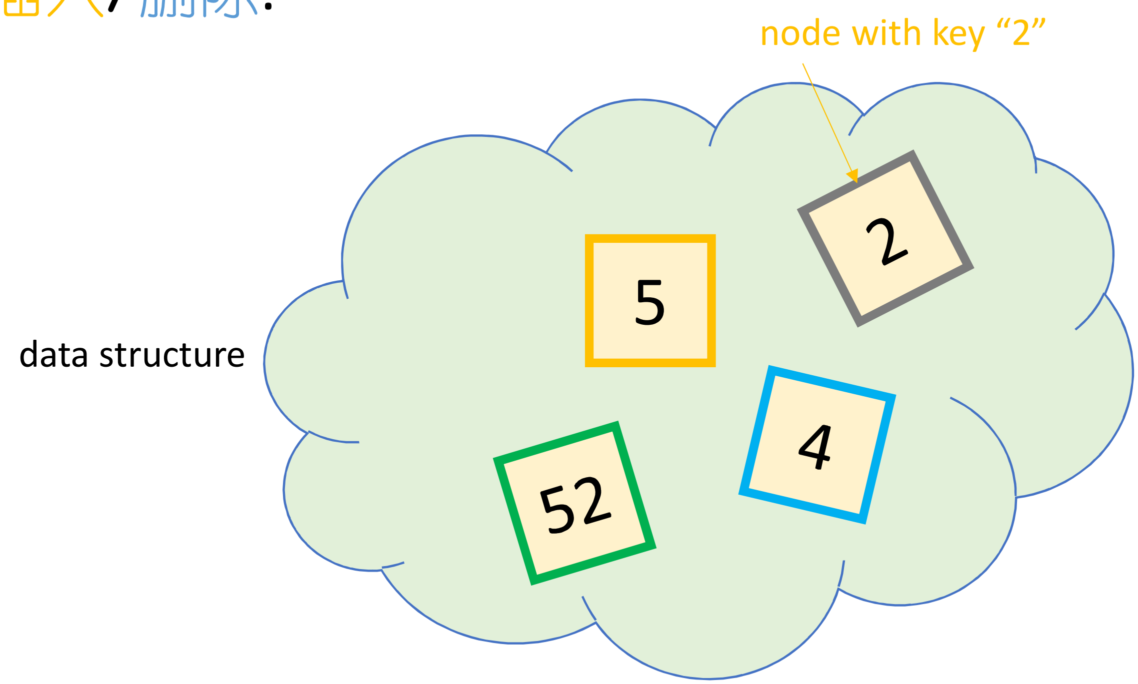
- Hashing

Hashing

哈希方法

回忆：数据结构要解决的问题

- 假设有几百万+数据需要维护，如何设计数据结构能够支持高效的查找/插入/删除.



The best of both worlds

	Sorted Arrays	Linked Lists	Balanced Binary 查找 Trees
查找	$O(\log(n))$	$O(n)$	$O(\log(n))$
插入/Delete	$O(n)$	$O(1)$	$O(\log(n))$

内容

- Hash表是这样的数据结构
  - 对比平衡二叉查找树

- 区别是这里需要用**randomness**在期望上达到O(1)，不过最差情况时会更差
  - Quicksort vs mergesort

- Hash families是实现的关键
- Universal hashing families更是实用上的进一步优化

Hashing

- CTO of Yahoo: “ 三种最重要的数据结构是 hashing, hashing and hashing.”
- Herbert Hellerman's *Digital Computer System Principles*, *1960s*
- 应用:
  - databases,
  - Compiler,
  - Computer security
  - 文件校验，Robin-Karp匹配算法
  - Java’s HashSet/HashMap, C++’s unordered_map

为什么要使用hashing

我们希望找到这样的数据结构，使得O(1)时间内实现查找/插入/删除

简单例子:

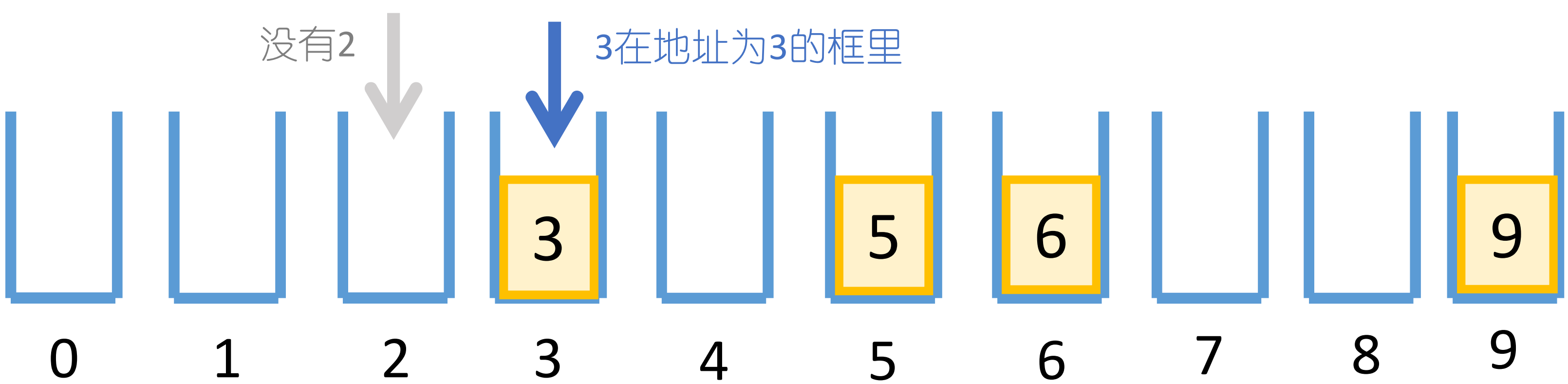
- 假如我们有n个不同的数，并且每个数都属于$[0,\ 2n)$，有没有可能在$O(1)$时间内查找/插入/删除某个X？

答： $bool\ *\ pData = new\ bool[2n];$

方法之一：直接寻址

即建立**n**个数的定义域到地址的一一映射

- 插入: 9 6 3 5
- 删除: 6
- 查找: 3 2



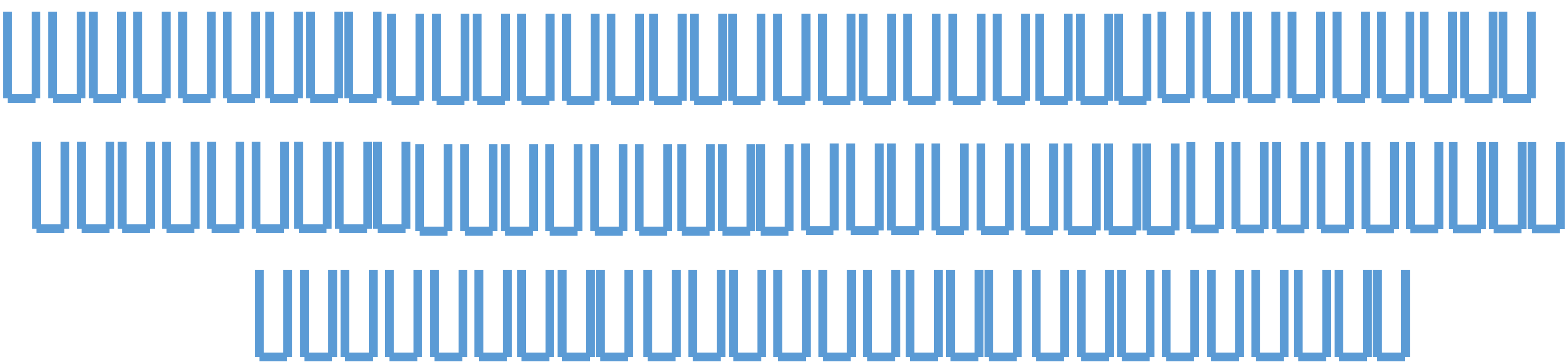
但是……,

- 同一问题，但是n个数是来自

$$U = \{0, 1, 2, \ldots\ldots, 10000000000\}$$

  比如数据的类型是64bit整型，即可以高达10^{19}

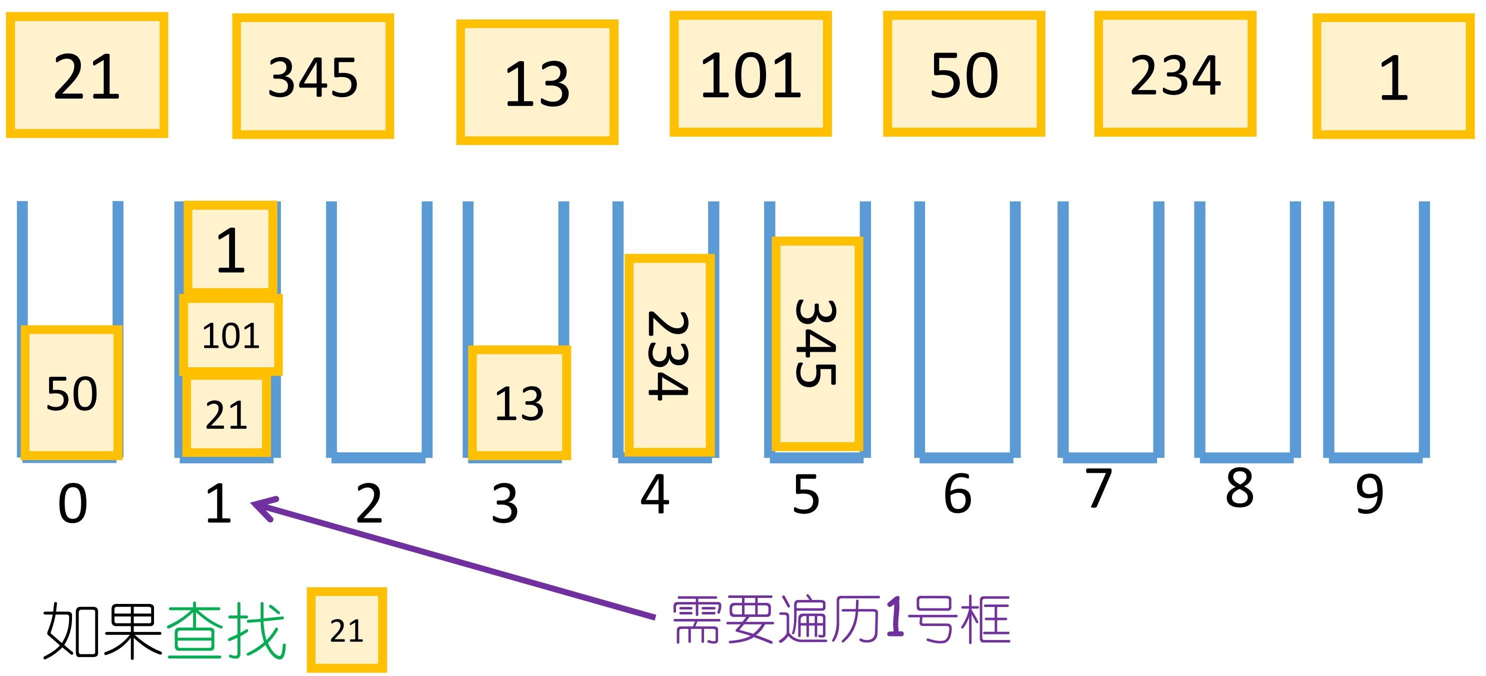
- 那需要10000000001个框来实现直接寻址



一种改进方法

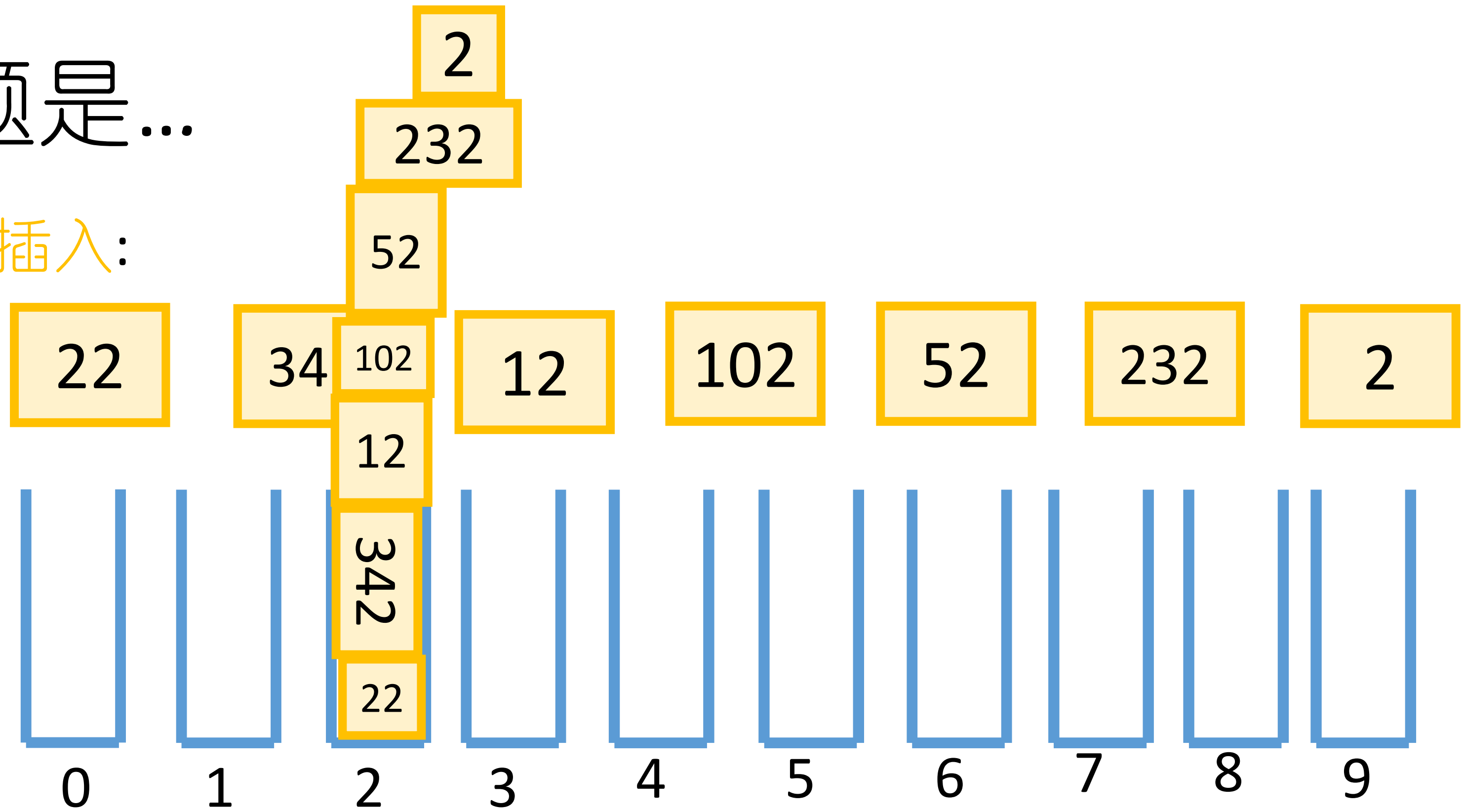
- 把n个数字按个位数放入10个框内

比如插入:



问题是...

插入:



如果查找 22

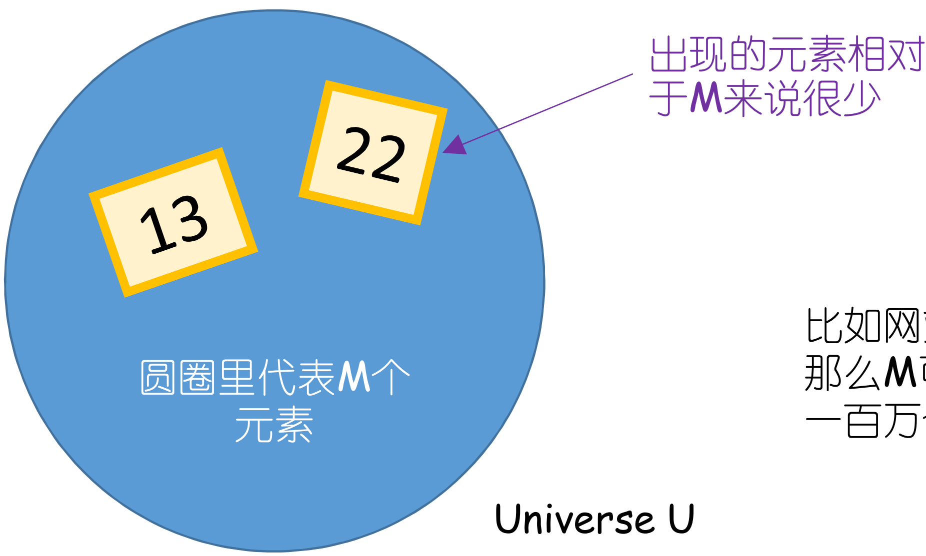
什么是Hashing？

什么是hashing

- 之前的例子就是，虽然效果很差
- 一般包括hash函数，hash表两部分
- 我们可以设计一个更好的……

术语介绍

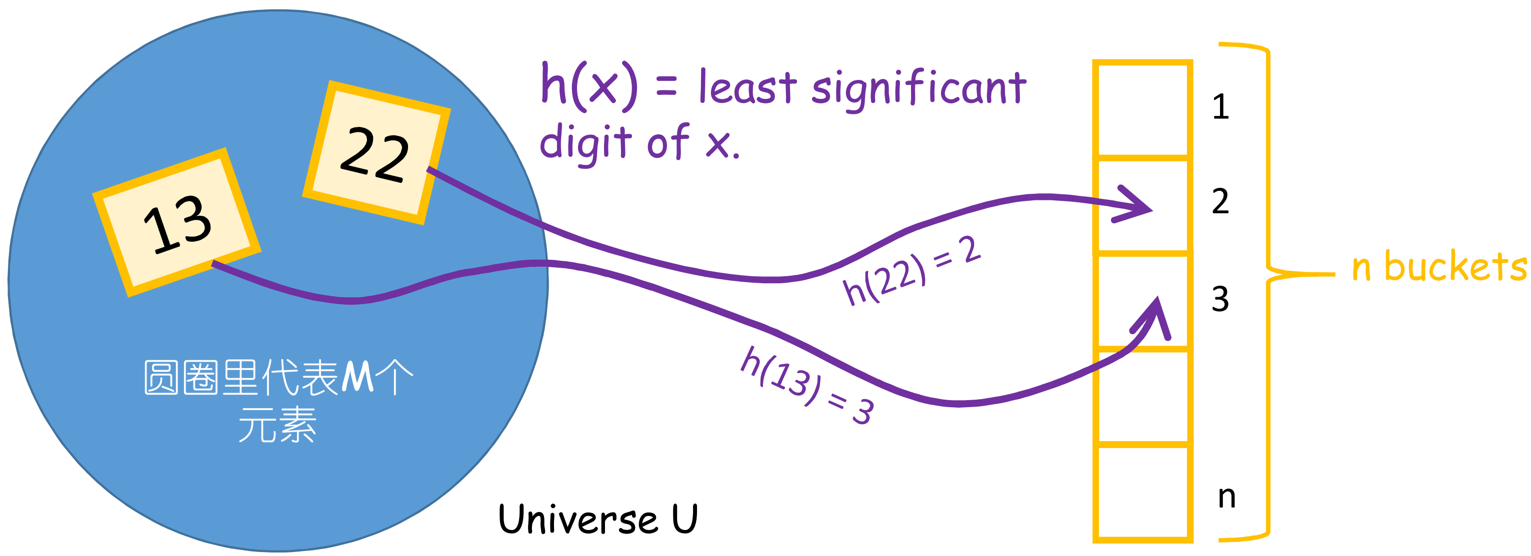
- 我们有很大的定义域 U， 数据元素个数为M.
  - M非常大，我们称为Universe.
- 只有其中**n**个数据元素可能被使用到.
  - M远远大于n.
- 不过我们不知道哪几个可能会被使用



比如网站的用户名可以是长度小于100的字符串，那么M可以取到128^{100}，但活跃用户可能只有一百万个，即$n=10^6$。

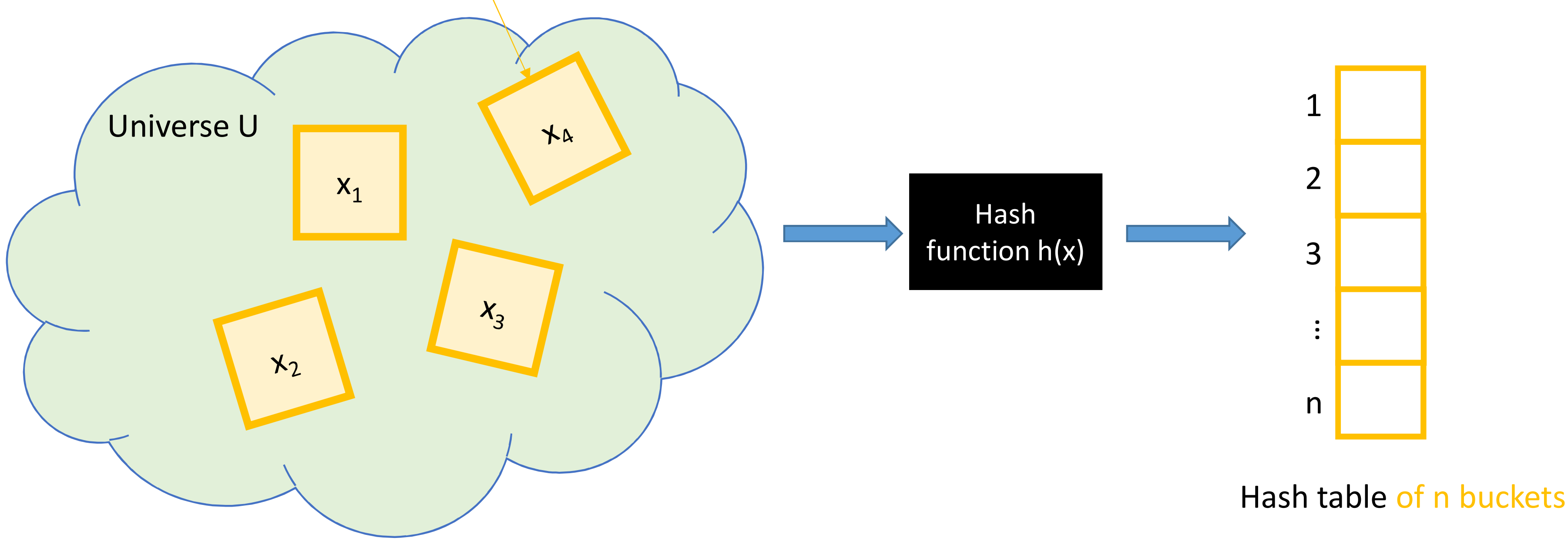
之前的例子

- 我们有很大的定义域 U， 数据元素个数为M.
  - M非常大，我们称为Universe
- 只有其中n个数据元素可能被使用到，且M>>n
- 我们只想用n个框，准备把n个用户数据放入
- 即，找这样的hash函数 $h:U \rightarrow \{1,\ldots,n\}$ 把元素映射到框地址



Hashing的思想

- 使用hash函数h(x)映射数据元素到hash table地址.
- 从而在Hash table（数组）中支持快速插入/删除/查找



Hash table

- 长度为$O(n)$的数组保存数据元素，因为要直接寻址
- 数组里的元素
  - 可以是数据本身，
  - 也可以是指向另一个数据结构的指针，比如链表，另一个 hash table，etc
- 在hash table上查找/插入/删除操作，甚至resize
- Hash table接近满时，需要resize，用load factor衡量

- 但是，哈希方法的好坏，很大程度上取决于一个好的 hash函数

什么是“好”的hash函数

“好”的hash函数，

- 要计算简单，
- 要把元素均匀映射到hash table各个地址
- 也就是任意$x \in U, i \in [1 \dots n]$,

$$\mathbf{Pr}[h(x) = i, h \in H] = \frac{1}{n}$$

这里$h \in H$是一个hash函数

Hash function例子

- 除留余数法
- 直接定址法
- 平方取中法
- 折叠法

- 理论上来讲，能真正随机的hash函数不可能是确定的某个函数，一种改进的做法是从所有“好”的hash函数里面随机选取一个
- 而这些符合要求的备选的hash函数，专称为hash family

首先一个问题是

- hash函数是压缩映射，即多对一映射
- 定义域U比n远远要大，肯定可以选到某些元素映射到同一个hash table地址
- 这个叫做hash冲突

先看个数学问题

Birthday paradox

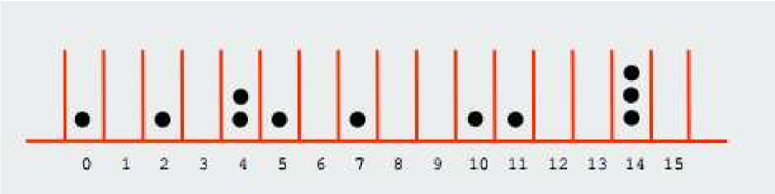
问：在一个房间里要多少人才能保证有两个人的生日一样的概率大于$1/2$？

- 假定每个人的生日独立，并且分布均匀

答：23个人能保证

概括为扔球进框的模型

- balls in bins
- 把m个球扔入n的框内
- 假定每次扔球独立，并且扔入任意一个框的概率一样
- 即扔入任意框内的概率都是$1/n$



- 问1：两个球落入同一个框内（冲突）的概率是多少？

扔球进框的模型

问2：扔m个球扔入n个框后，冲突的平均次数是多少？

如果用X_{ij}表示第i个球和第j个球冲突的事件（都到同一个框里，概率为$1/n$）。
令当冲突事件发生时$X_{ij} = 1$冲突事件不发生的时候$X_{ij} = 0$

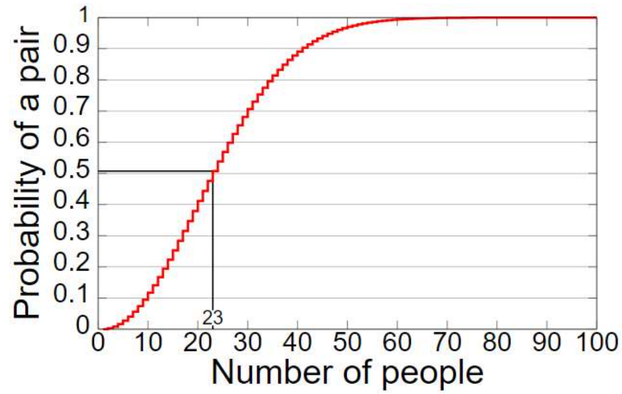
那么$X = \sum_{i \neq j} X_{ij}$就是冲突次数，所以

$$\mathbf{E}[X] = \sum_{i \neq j} \mathbf{E}[X_{ij}]$$

$$= \sum_{i \neq j} \mathbf{Pr}[X_{ij} = 1] = \frac{1}{n}\binom{m}{2}$$

扔球进框的模型

- Birthday paradox就是要扔多少球才可能存在某一框内有两个球（至少1次冲突）的概率大于1/2？



- 人 $\Leftrightarrow m$个球
- 一年天数 $\Leftrightarrow n$个框
- 计算 $\mathbf{E}[X]=\frac{1}{365}\binom{m}{2}=1$ 得
- $m \geq 27$，即扔27个球后，平均冲突次数达到1，不过这个是平均值
- 反面是计算没有冲突得概率：$(1-1/n)^{\binom{m}{2}} \leq 1/2$，得$m \geq 23$

扔球进框的模型

问3：要扔多少个球才能使得每个框内都至少有一个球？

首先第i个框内是空的概率是$\left(1-\frac{1}{n}\right)^m \approx e^{-m/n}$

那么空框的期望个数是$\sum_i 1 \times \Pr(i-\mathrm{th}框是空的) = n \times e^{-m/n}$

当$m = O(n\mathrm{log}n)$时，上式< 1

扔球进框的模型

问4：扔n个球后，最满的框内有几个球，只要求在概率趋于1情况下？

- 因为第i个框有至少k个的球的概率是

$$\binom{n}{k}\left(\frac{1}{n}\right)^k \le \frac{n^k}{k!}\cdot\frac{1}{n^k} \le \frac{1}{k!} \le 1/k^{k/2}$$

- 当$k^* \ge \frac{8\log n}{\log\log n}$，可以证明上式$\le 1/n^2$

- 那么存在一个框达到$k^* \ge \frac{8\log n}{\log\mathrm{lo}}$的概率是$n \times 1/n^2 = 1/n$,

- 反面就是，最满的框里只有$O\left(\frac{\log n}{\log\log n}\right)$个球的概率是$(1-\frac{1}{n}) \to 1$

回到hash冲突

hash冲突

- 就是两个元素映射到同一个hash table地址
- Birthday paradox意味着冲突肯定存在，除非n非常非常大
- 关键是如何有效的解决冲突
  - 闭地址法
    - 拉链法
    - Perfect hashing，在输入数据是给定的情况下可以实现的
  - 开地址法
    - ***Linear probing***
    - ***Quadratic Probing***
    - ***Double hashing***

1. 拉链法

- 对于任意一个Hash function $h: U \rightarrow \{1,\dots,n\}$
- Hash table是一个长度为n的数组,
- 数组里每个元素都是指向链表的头节点
- 然后所有映射到同一个地址的元素都插入到对应的链表里面

1. 拉链法

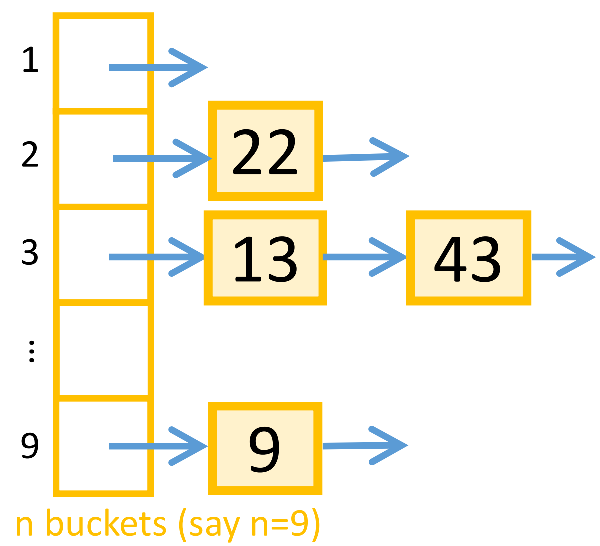
比如我们取Hash function h(x) = least significant digit of x

冲突的元素都插入到同一个链表里.

- 插入链表的复杂度是O(1)
- 不过查找的复杂度是O(链表长度).

插入:

查找 43: 计算h(43) = 3，再遍历对应链表



1.拉链法

- 如果我们有一个“好”的hash函数，使得映射到每个框里的概率一样
- 根据之前的balls in bins模型，最多有$O(\log n\ /\ \log\log n)$ 个球的概率几乎为1
- 再改进一下，如果用两个这样的好hash function，每次加入新元素的时候放在短的链表上，那么链表最大长度可以改进为$O(\log\log n)$

开放定址法

- 在一次hashing后发现冲突后，
- 继续探查下一个空的位置，直到找到为止
- 也就是，我们把hash函数扩展到

$$h : U \times \{0, 1, \ldots, n-1\} \rightarrow \{1, \ldots, n\}$$

这样，我们探查的位置依次是$h(x, 0), h(x, 1), \ldots, h(x, n-1)$

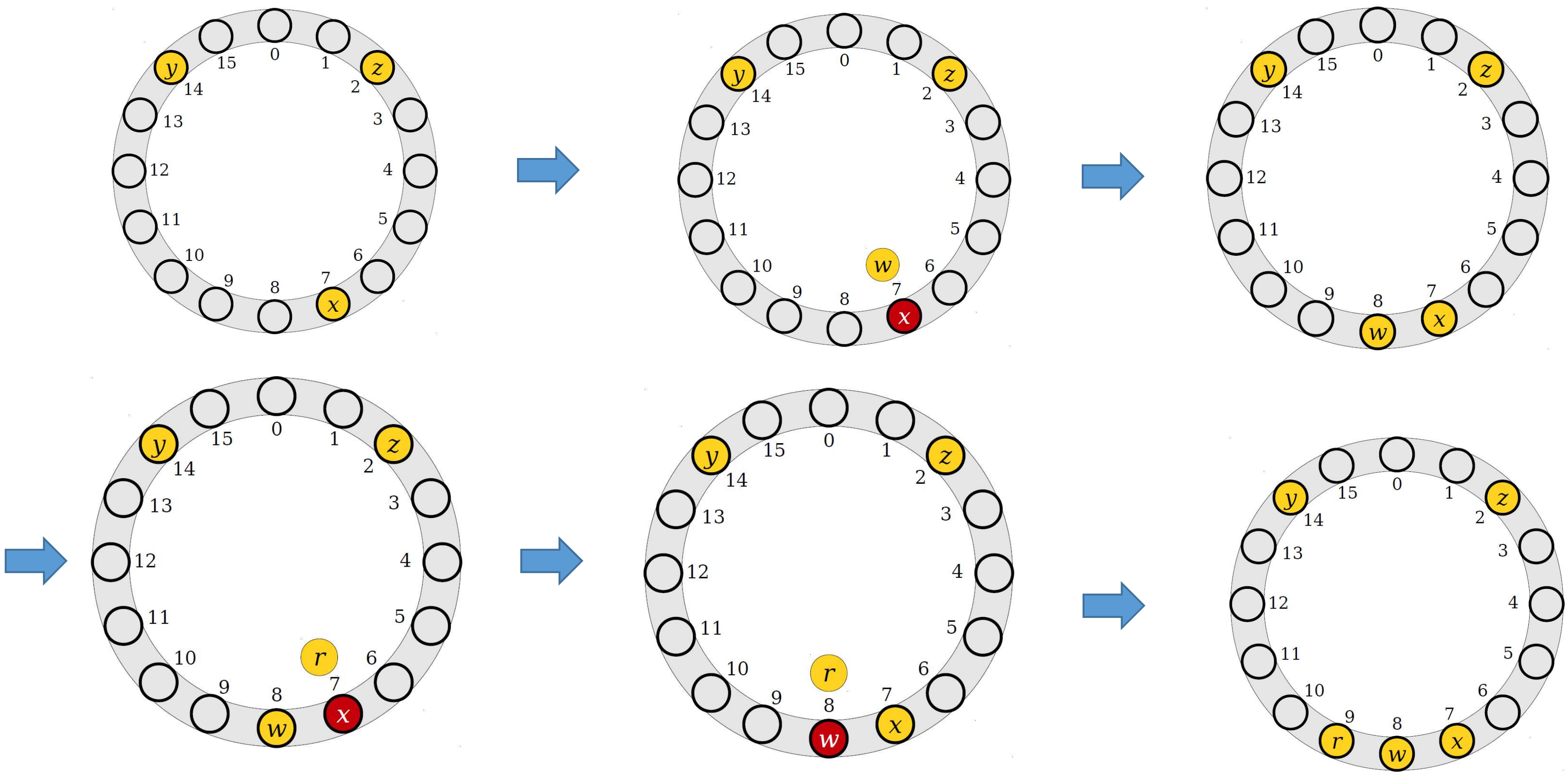
注意，依次探查的位置最好是$\{1, \ldots, n\}$的一个排列，这样我们就能插入新元素，只要有空位的话

2.1 linear probing

- LinearProbing的hash函数是$h(x,i) = (h'(x) + i) \ mod \ n$
- 这里$h'(x)$ 是正常的hash函数
- 如果$h'(x)$ 被占用，那么就依次往后移动一位

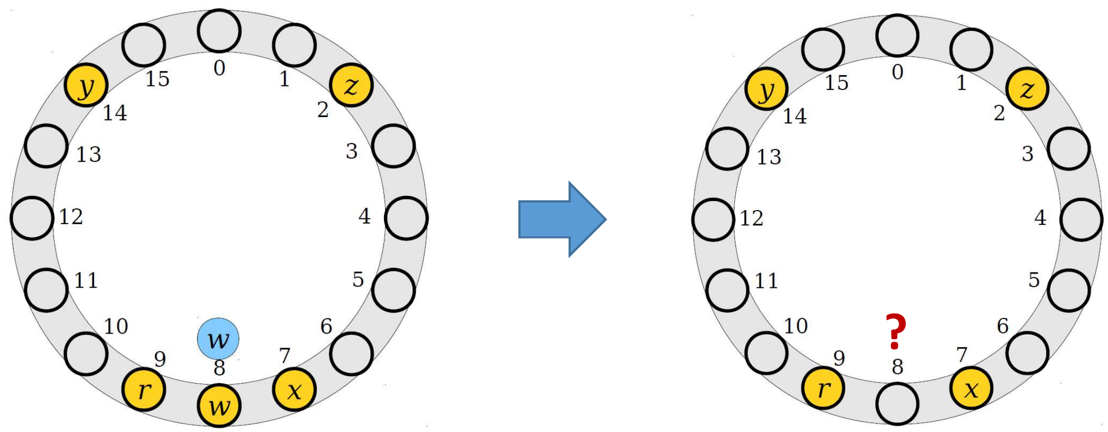
- 查找：
  - 根据$h'(x)$计算地址，然后逐次找下一个
  - 直到找到，或者是空的元素

2.1 linear probing插入

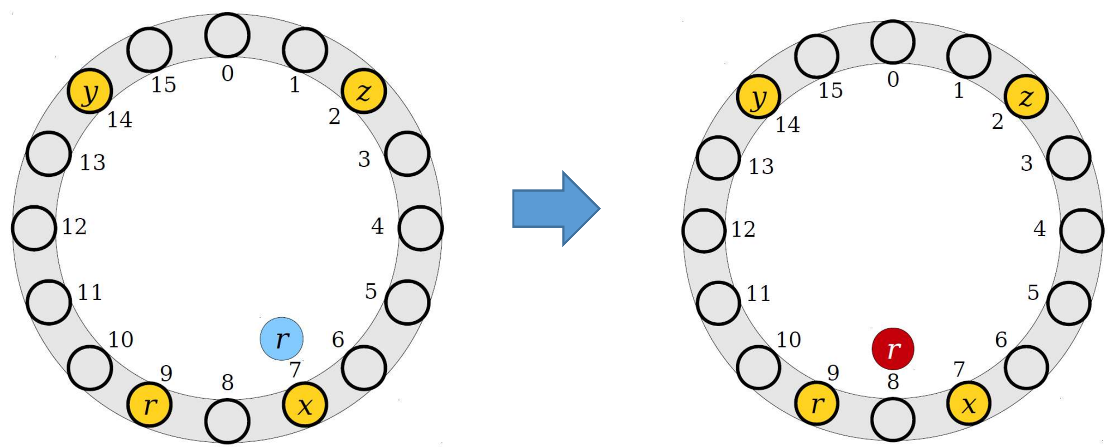


2.1 linear probing删除

- 先查找到要删除的元素
- 但是如果直接删除会有问题

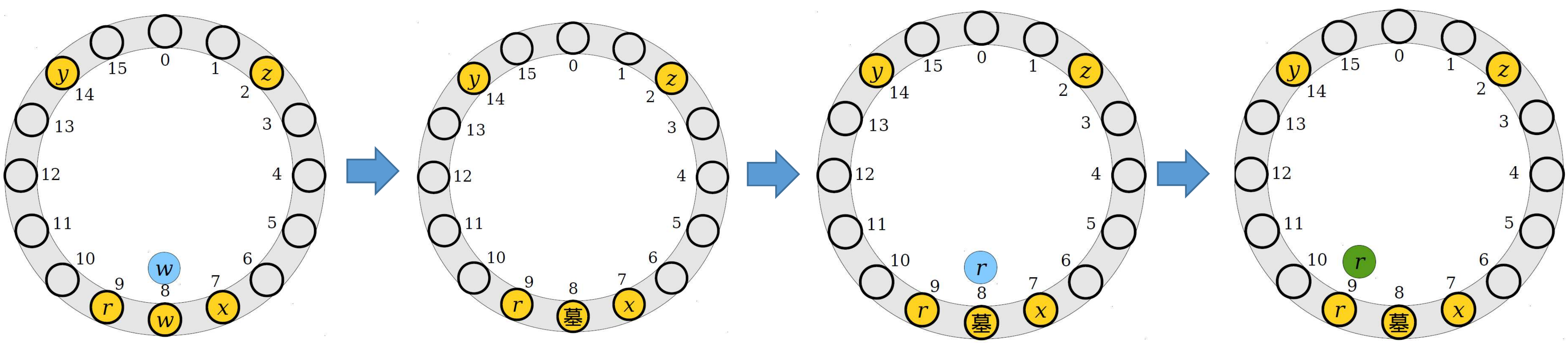


- 比如，删除w后，再查找r



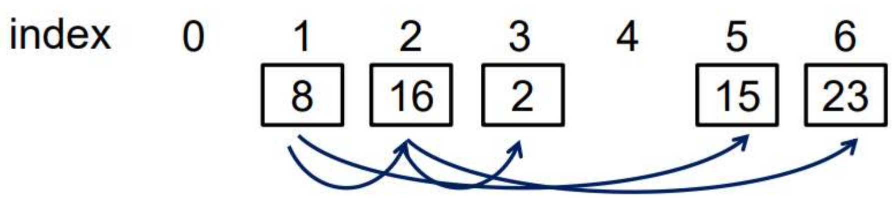
2.1 linear probing删除

- 不要直接删除元素，而是设个tomb标志
- 之后如果查找，就跳过设有删除标志的元素
- 如果插入，可以直接插入到已经删除的位置



2.2 Quadratic probing

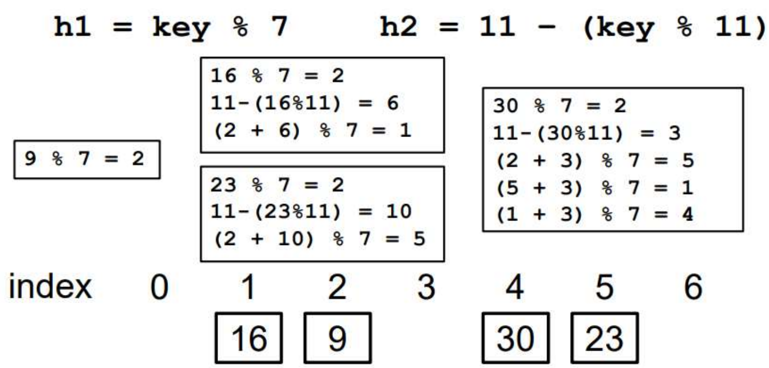
- QuadraticProbing的hash函数是
$$h(x,i) \;=\; (h'(x) \;+ c_1 i + c_2 i^2)\; mod\; n$$
- 这里$h'(x)$ 是正常的hash函数
- 比linear probing效果要好
- 但是要使得每个可能位置都取到，对c_1， c_2，和n都有限制
- 比如$n = 7$，插入8， 16， 15， 2，23
- 使用$h(x,i) = (x + i^2)\; mod\; 7$



2.3 double hashing

- DoubleHashing的hash函数是$h(x,i) = (h_1(x) + ih_2(x))\ mod\ n$
  - 这里$h_1(x)$ 和$h_2(x)$是正常的hash函数
  - $h_2(x)$不要等于0

- 比如对$n = 7$，插入数是9，16，23，30



Perfect Hashing

什么是完美hashing

- 首先，hash table长度是插入元素个数的常数倍
- 其次，每个地址冲突的次数是常数次

- 也就是，即是空间利用率最优
- 同时保障了操作时间上的最优，即使在最差情形

- 要达到这样的效果，这里要求插入元素集合是静态的，即给定的
- 比如刻录在光盘上的数据，编程语言的关键词，等等

如果不考虑空间，怎么解决冲突

定理：如果插入元素集合有n个，那么对于一个“好”的hash 函数，可以在大小为n^2的hash table上实现冲突期望次数<1

回忆balls in bins，扔m个球扔入n个框后，冲突的期望次数是

$$\frac{1}{n}\binom{m}{2}$$

如果$n = m^2$，那么上式$< 1/2$

也就是说冲突的次数是常数次

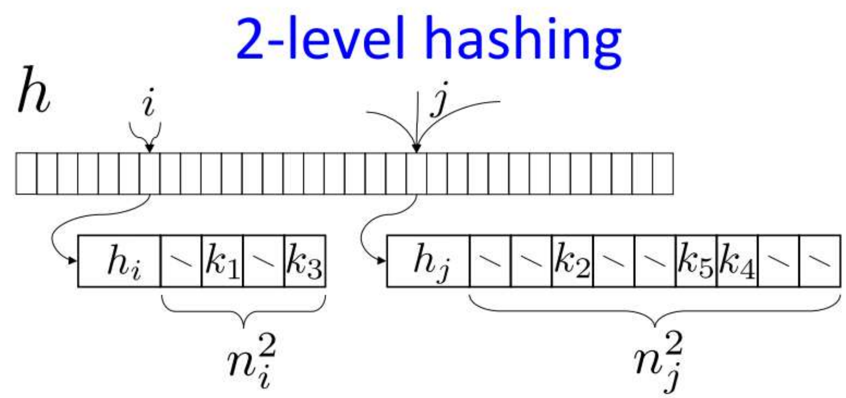
完美hashing

- 进一步，在保证不冲突情况下，能不能保证线性空间？
- 也就是完美hashing

- 方法是先对n个元素在线性$O(n)$大小的hash table上hashing一次，
- 然后对冲突的元素，再进行一次hashing，不过子hash table空间给足够多，也就是平方倍的，使得第二次hashing后没有冲突

完美hashing证明

- 假如在第一次hashing后，第i个地址上有n_i个元素，
- 那么对这些元素创建大小为$n_i{}^2$的子hash table，再进行一次hashing（可以用不同的好的hash function）
- 那么$n = \sum_i n_i$,需要额外空间$\sum_i n_i{}^2$，可以证明这个额外空间是线性的
- 因为$\sum_i n_i{}^2 = \sum_i n_i + 2\sum_i \binom{n_i}{2}$
- 前者是n，后者等于所有冲突次数，根据balls in bins模型第二问，得出$= 2\frac{1}{n}\binom{n}{2} = n - 1$



为什么要完美hashing

- 对于给定的n个数据元素，有没有更直接的方法？
  - 如果给定的数据都是$[0, 2n)$，能直接寻址
  - 如果给定的数据是n个用户名，寻址映射不是那么容易
  - 任意数据的话……

  - 完美hashing的方法提供了一种简单的寻址映射的机制

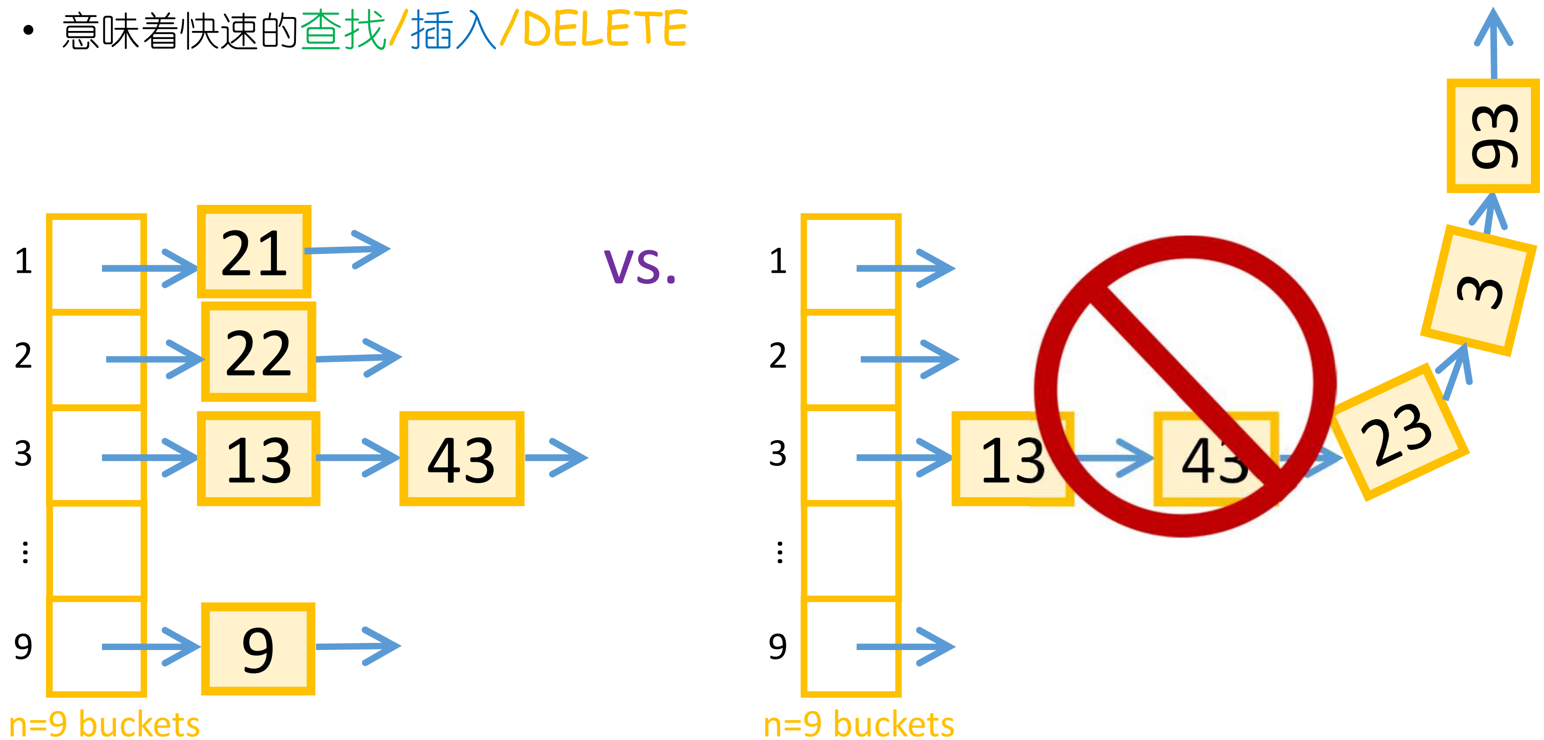
以上是处理Hash table的方法，

Hashing最主要的问题……

主要问题

怎么选取一个“好”的hash函数？

1. 从空间上的考虑，不能有太多的bucket (比如n)
2. 元素要在尽可能的分散，减少冲突。即每个bucket被映射到的概率差不多
   - 意味着快速的查找/插入/DELETE

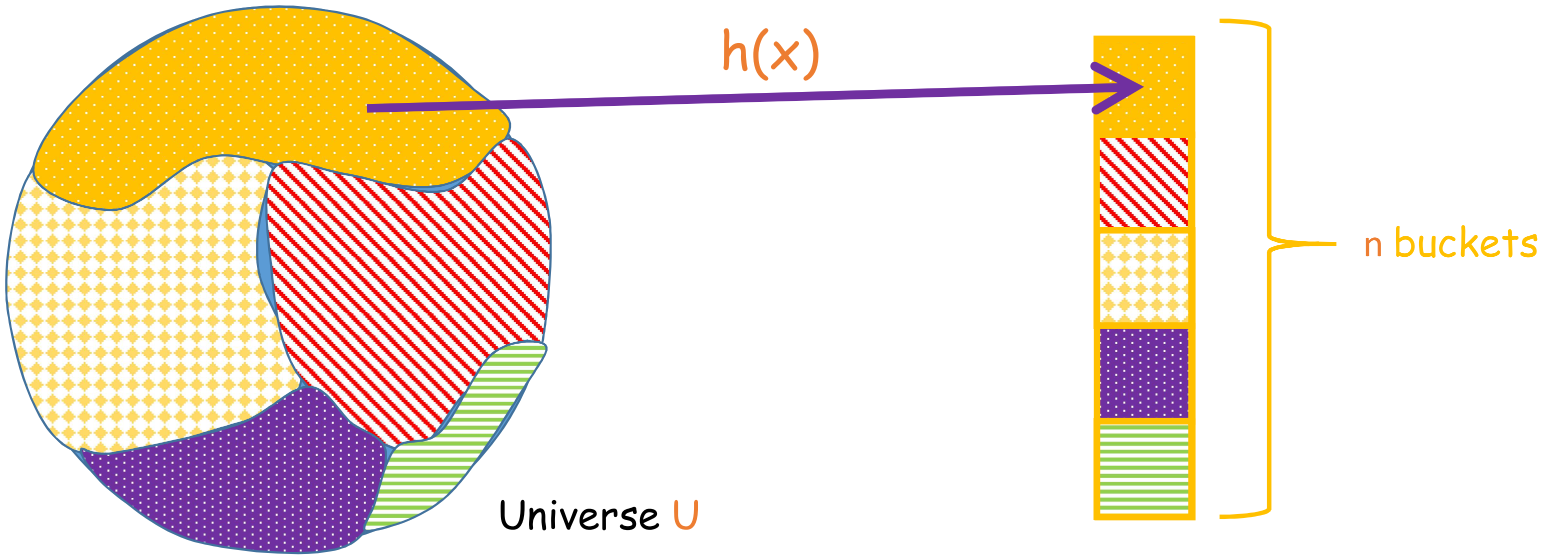


问：

- 有没有这样一个函数h: U -> {1,...,n} 使得:
  - 无论输入任何可能的O(n)个元素
  - 都能保证基本没有冲突

- 如果我们能找到这样的函数，那么也就实现了希望的 O(1) 插入/删除/查找

No 对任意一个确定的函数h(x)



因为$|U| = M$远远大于n，根据鸽笼原理，至少有M/n的元素会被选定的函数映射到同一个地址

结论是最差情况一定存在，特别如果用户数据有bias，可能性还不小

Solution: Randomness

用户数据可能有bias，hash函数变随机

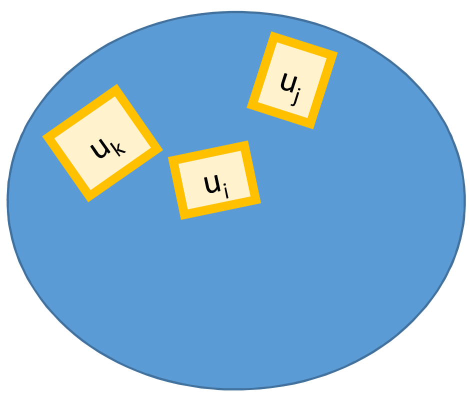
如何加入随机因素：

1. 我们找一批合适的hash函数组成集合（就是后面要讲的universal hash family）
2. 从这批函数集合中随机选出一个来，进行hashing

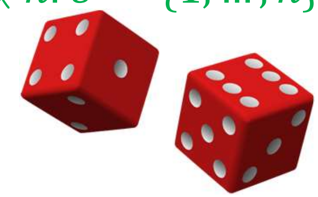
hashing随机策略

1. 你的对手可以随意选择n个元素 $u_1, u_2, \ldots, u_n \in U$，可以执行任意的查找/删除/插入操作组合

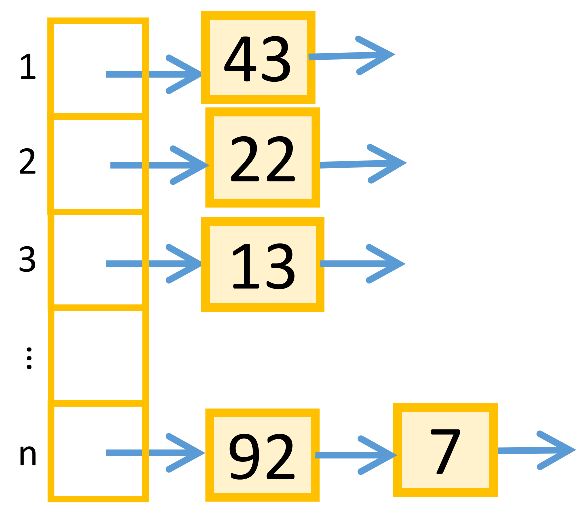
插入 13，插入 22，插入 43，插入 92，插入 7，查找 43，删除92，查找 7，插入 92



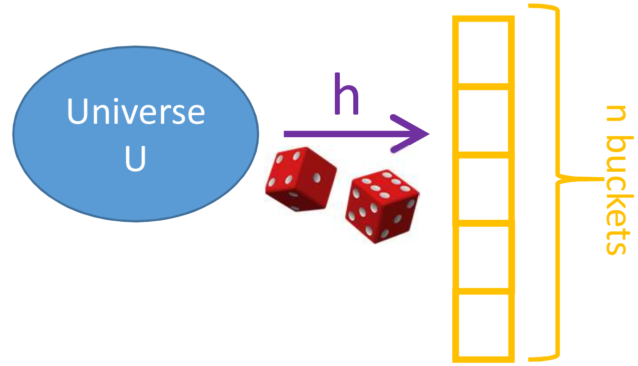
2. 算法的策略是，不管对手怎么样，我们都选出一个随机的hash函数 $h: U \to \{1, \ldots, n\}$.



3. 进行hash操作



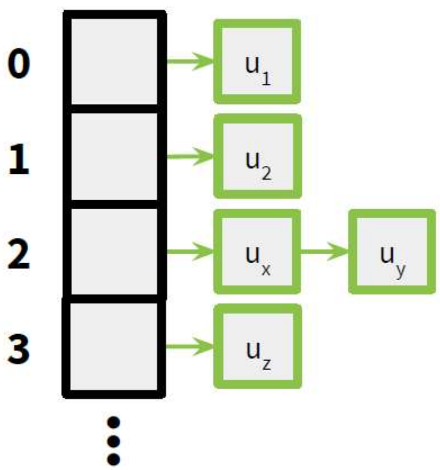
这个策略为什么有用

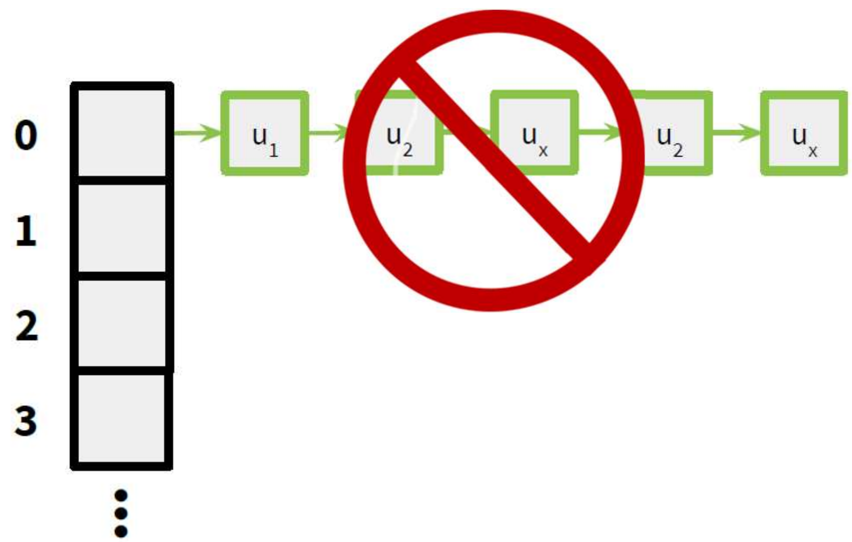


- 如果选出来的**h**是真正随机的
  - 意味着**$h(x_1)$**是在**1**到**n**之间的随机数
  - 意味着**$h(x_2)$**是在**1**到**n**之间的随机数，且独立于**$h(x_1)$**
  - 意味着**$h(x_3)$**是在**1**到**n**之间的随机数，且独立于**$h(x_1)$**，**$h(x_2)$**
  - …
  - **$h(x_n)$**是在**1**到**n**之间的随机数，且独立于**$h(x_1)$**，**$h(x_2)$**，……，**$h(x_{n-1})$**

希望策略能达到的效果

- 我们不希望出现右边的情况，就是u_x所在的地址有很多冲突
- 我们希望对于任意u_x，u_x所在的地址的冲突都是$O(1)$





hash随机算法

- 设计这样一个hash函数的集合H（hash family），我们从中随机选取一个hash函数，在对手任意选定n个元素后，使得对于每个u_x，我们都能保证，u_x所在的框的冲突的期望值是$O(1)$

- 注意，这里不能用所有框的期望值是$O(1)$代替

好消息

- 一个简单的设计就是把所有的可能的映射都放到H这个hash family集合里面，那么总共有$|H| = n^{|U|}$个不同的映射

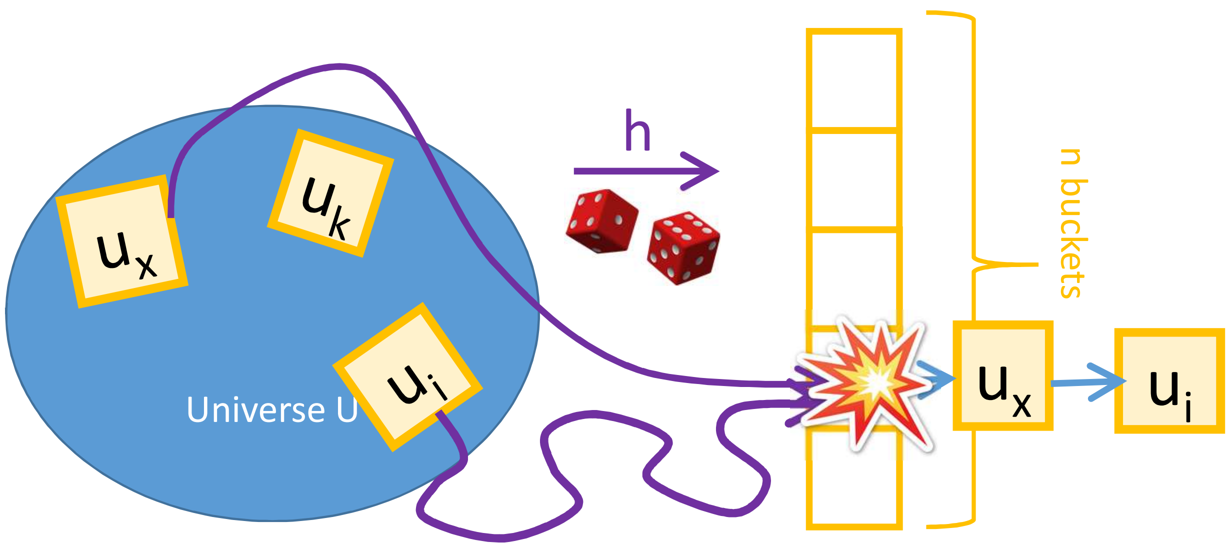
- 比如定义域universe是{ “a”， “b”， “c” }，映射到的框（地址空间）只是{0， 1}

	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8
“a”	0	0	0	0	1	1	1	1
“b”	0	0	1	1	0	0	1	1
“c”	0	1	0	1	0	1	0	1

这里h_8把“b”映射到框1

Expected number of items in u_x's bucket?

- $E[\ ] = \sum_{i=1}^{n} P\{\, h(u_x) = h(u_i)\}$
- $\quad = 1 + \sum_{i \neq x} P\{\, h(u_x) = h(u_i)\}$
- $\quad = 1 + \sum_{i \neq x} 1/n$
- $\quad = 1 + \frac{n-1}{n} \leq 2.$



坏消息

- 穷举的集合太大$n^{|U|}$，不比直接寻址需要的空间少
- 我们想找规模小的hash family，使得从里面取出的hash函数还是接近于真正随机
- 这样即能保证期望意义上的常数时间内的操作
- 又使得保存hash family的空间不用太大

问题等价于

- 设计一个规模大小可以接受的hash函数集合H，使得从中随机选取一个hash函数h，满足

$$\text{for all } u_x, u_i \in U \quad \text{with } u_x \neq u_i,$$
$$P_{h\in H}\{\, h(u_x) = h(u_i)\} \leq \frac{1}{n}$$

- 这样我们还是能得到操作的期望复杂度是O(1)

- 这样的hash函数集合，我们专门叫做 universal hash family

universal hash family著名例子

- 选一个素数$p \geq M$.
- 定义

$$f_{a,b}(x) = ax + b \quad mod\ p$$

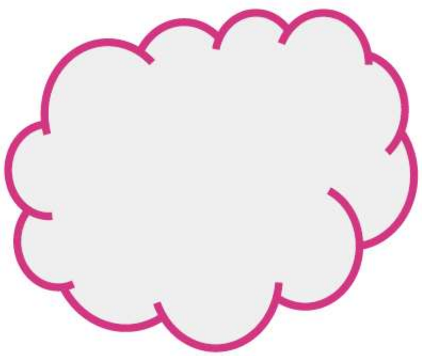
$$h_{a,b}(x) = f_{a,b}(x) \quad mod\ n$$

- 那么

$$H = \{\, h_{a,b}(x) \;:\; a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\} \,\}$$

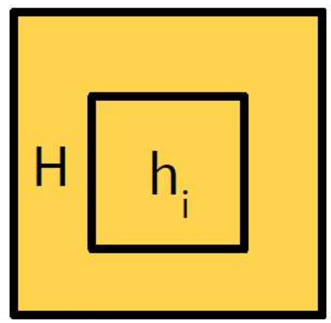
就是一个universal hash family

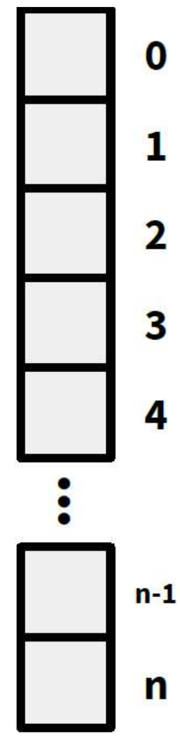
总结



1. 你先选定一个universal hash family, 比如 H = mod p mod n

2. 用户开始初始化hash table, 你随机选一个h_i，之后用h_i来hashing用户的输入

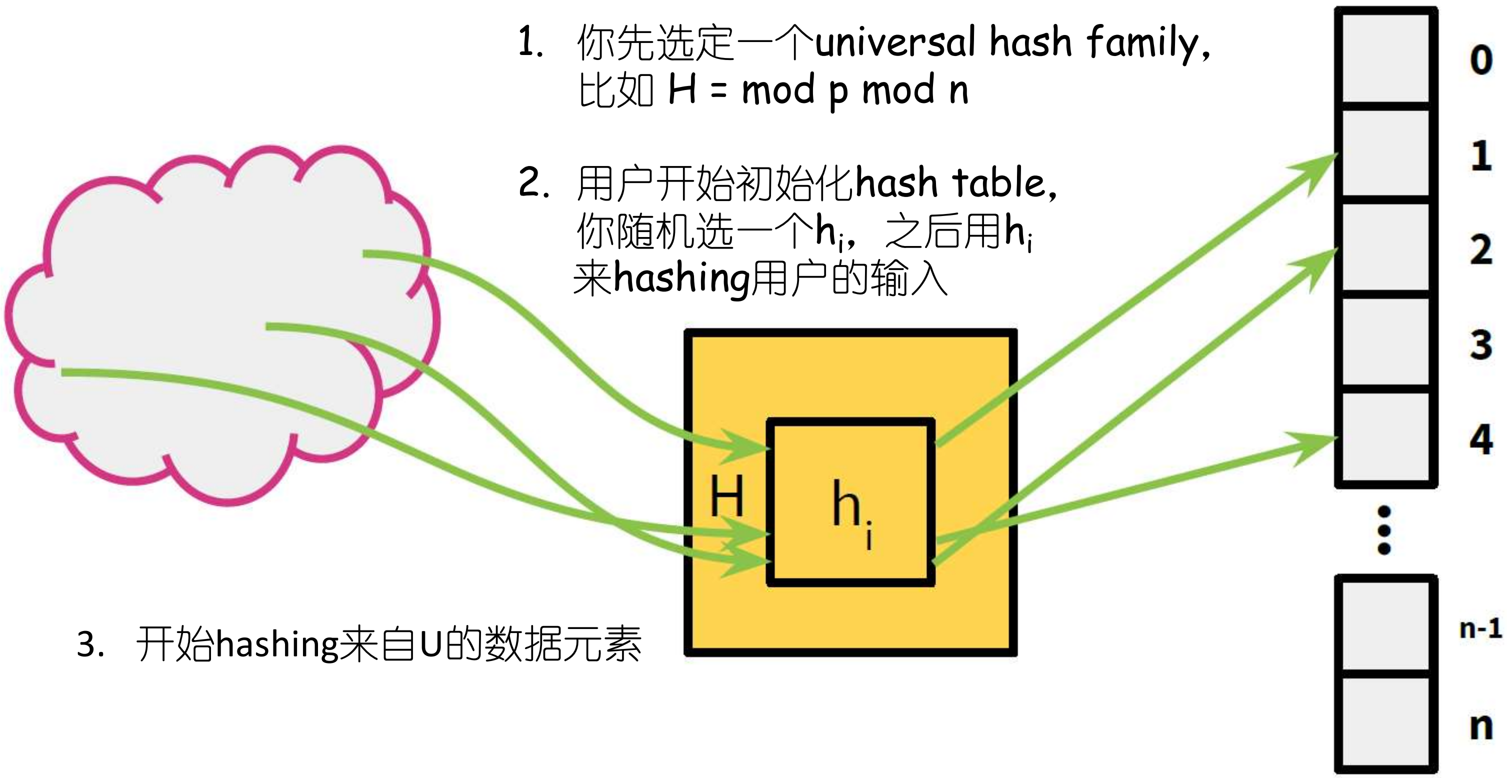




总结

1. 你先选定一个universal hash family，比如 H = mod p mod n

2. 用户开始初始化hash table，你随机选一个h_i，之后用h_i来hashing用户的输入



3. 开始hashing来自U的数据元素

最后

- 和quicksort一样，存在最差情况。
- 实际中用不同的策略换hash function
  1. 一有冲突，就再随机选个新的hash function
  2. 等冲突足够多了，再随机选择
- 如果换了新的hash function，这时候需要
  - 对已有的数据用新的hash函数rehash
  - 新开一个hash table，同时保留之前的
    - 新的用新的hash function，插入都在新的hash table
    - 旧的还用旧的hash function，只用来查找/删除
    - 在查找的时候可以转移到新的hash table

The best of both worlds

time	Sorted Arrays	Linked Lists	Balanced Binary 查找 Trees	Hash table
查找	O(log(n))	O(n)	O(log(n))	O(1)
插入/Delete	O(n)	O(1)	O(log(n))	O(1)

检查树平衡

1. Top down

```
int depth (TreeNode *root)
{
    if (!root) return 0;
    return max (depth(root->left), depth (root->right)) + 1;
}

bool isBalanced (TreeNode *root)
{
    if (!root) return true;

    int left=depth(root->left);
    int right=depth(root->right);

    return abs(left - right) <= 1 &&
           isBalanced(root->left) &&
           isBalanced(root->right);
}
```

2. bottom up

```
int dfsHeight (TreeNode *root)
{
    if (!root) return 0;

    int leftHeight = dfsHeight(root->left);
    if (leftHeight == -1) return -1;

    int rightHeight = dfsHeight(root->right);
    if (rightHeight == -1) return -1;

    if (abs(leftHeight - rightHeight) > 1)  return -1;
    return max (leftHeight, rightHeight) + 1;
}
bool isBalanced(TreeNode *root)
{
    return dfsHeight (root) != -1;
}
```

树的轮廓

- 输出二叉树的轮廓
  - 左边界（从上到下）
  - 右边界（从下到上）
  - 叶节点（左子树，右子树）
  - 避免重复输出

```
void printLeaves(struct node* root)
{
    if (!root)
    {
        return;
    }

    printLeaves(root->left);

    // Print it if it is a leaf node
    if (!(root->left) && !(root->right))
    {
        cout << root->data << " ";
    }

    printLeaves(root->right);
}
```

```
void printBoundary (struct node* root)
{
    if (!root)
    {
        return;
    }

    cout << root->data << " ";

    // Print the left boundary in top-down manner.
    printBoundaryLeft(root->left);

    // Print all leaf nodes
    printLeaves(root->left);
    printLeaves(root->right);

    // Print the right boundary in bottom-up manner
    printBoundaryRight(root->right);
}
```

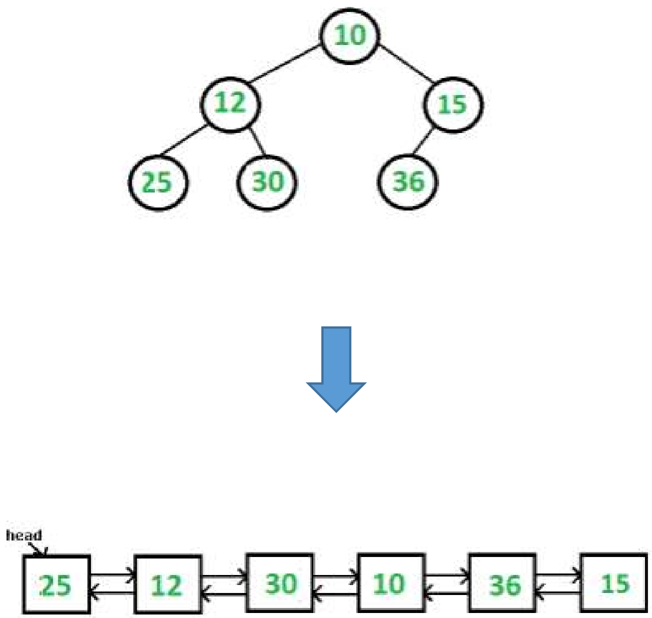
左右边界

```cpp
void printBoundaryLeft(struct node* root)
{
    if (!root)
    {
        return;
    }

    if (root->left)
    {
        // to ensure top down order, print the node
        // before calling itself for left subtree
        cout << root->data << " ";
        printBoundaryLeft(root->left);
    }
    else if(root->right)
    {
        cout << root->data << " ";
        printBoundaryLeft(root->right);
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}
```

```cpp
void printBoundaryRight(struct node* root)
{
    if (!root)
    {
        return;
    }

    if (root->right)
    {
        // to ensure bottom up order, first call for right
        //  subtree, then print this node
        printBoundaryRight(root->right);
        cout << root->data << " ";
    }
    else if (root->left)
    {
        printBoundaryRight(root->left);
        cout << root->data << " ";
    }
   // do nothing if it is a leaf node, this way we avoid
   // duplicates in output
}
```

树转换成双向链表

```
void treeToDoublyList(Node *root, Node **head)
{
  if (!root) return;

  static Node *prev = nullptr;

  // Recursively convert left subtree
  treeToDoublyList(p->left, head);

  // Now convert this node
  if (!prev)
  {
    *head = root;
  }
  else
  {
    root->left = prev;
    prev->right = root;
  }

  // updates previous node
  prev = root;

  treeToDoublyList(root->right, head);
}
```



作业

- 在BinaryTree实现下面两个成员函数，并测试通过：

- 把二叉树按中序转换成双向链表： TreeToDoubleList(LinkList * out)
  - 递归实现
  - 不能使用static 变量
  - 输出是个双链表类的对象
  - 树里的节点要释放掉，即最后要清空树节点，都换到双链表里

- 找二叉树最深的左叶节点
  - 递归实现
  - 比如： void DeepestLeftLeaf(int lvl, bool isleft, int *maxDepth, Node *pLeaf)， pLeaf返回找到的叶节点指针，maxDepth返回当前找到的最深左叶节点的深度，用指针即作为输入也作为输出参数

Q&A

Thanks!