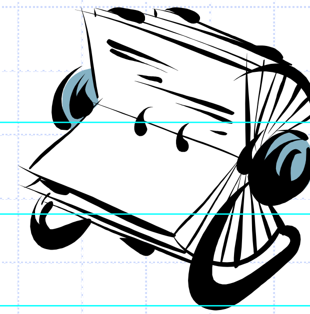
Maps

Maps



- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed [key = name value = address]
- Applications:
  - address book
  - student-record database → [key = roll no, value = student data]

$\{k_1 \Rightarrow v_1, k_2 \Rightarrow v_2 \ldots k_n \Rightarrow v_n\}$

The Map ADT (§ 8.1)

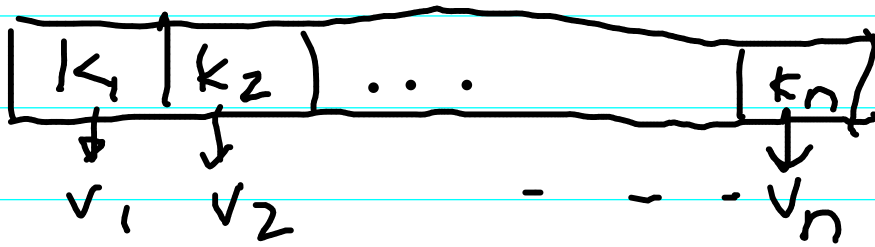
- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k & replace the value
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - keys(): return an iterator of the keys in M → No repetitions
  - values(): return an iterator of the values in M → can have repetitions

{1500 ⇒ "Bombay"} .put(1500, "Mumbai")
will replace "Bombay" with "Mumbai" & return "Bombay"

Q: Implementation(s) of Map that take less time for get(k) while being "reasonable" for put(k,v)

Ideas: (a) Assuming a way of sorting keys

$k_1 < k_2 < \dots k_n$



get(k) through binary search in $O(\log(n))$

put(k,v) while respecting sorting will be $O(n)$ [even with amortised analysis]

Eg:
```
public Class Pair {
    String key;
    String value;
}
```

could implement as an array Pair[] but maintain sorting on key field

(b) Linked list of Pair:

get(k) : $O(n)$

put(k, v) : $O(1)$

$O(n)$ for a "faithful" implementation that "gets" ⟨k, v⟩ if k was already present

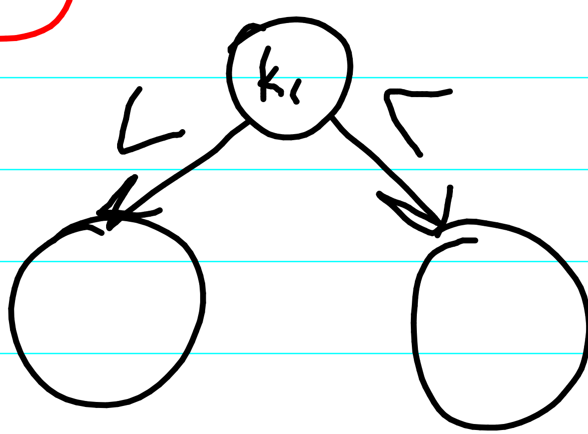
Assuming insertion at beginning so that get(k) gets last inserted value scanning from "front"

(c) Encoding function on key to map "key" to a "position" in an array

Q: Is this a "unique" position?

can $f(k_1) = f(k_2)$ for $k_1 \neq k_2$ [i.e collision for some k_1, k_2]?

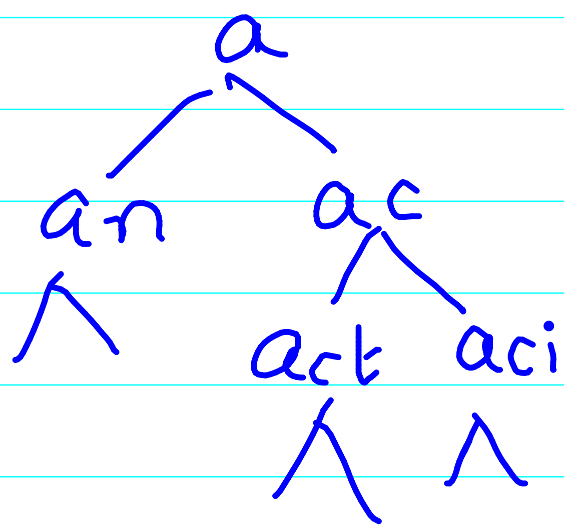
(d) Binary search tree



get(k): $O(\log n)$

put(k, v) : $O(\log(n))$

(e) Trie (prefix trie)



(f) Assuming key length $\leq r$

Take an "r" dimensional array

More elaboration on c:

1. Looks difficult to come up with a "hash" function f that will give no collisions.

Let us consider $f(\text{"ant"})$ = ASCII value of first character = 65

would like to avoid growing

a Choice of "hash" functions

a Should give close to "uniform" distribution over all keys for the buckets

Example

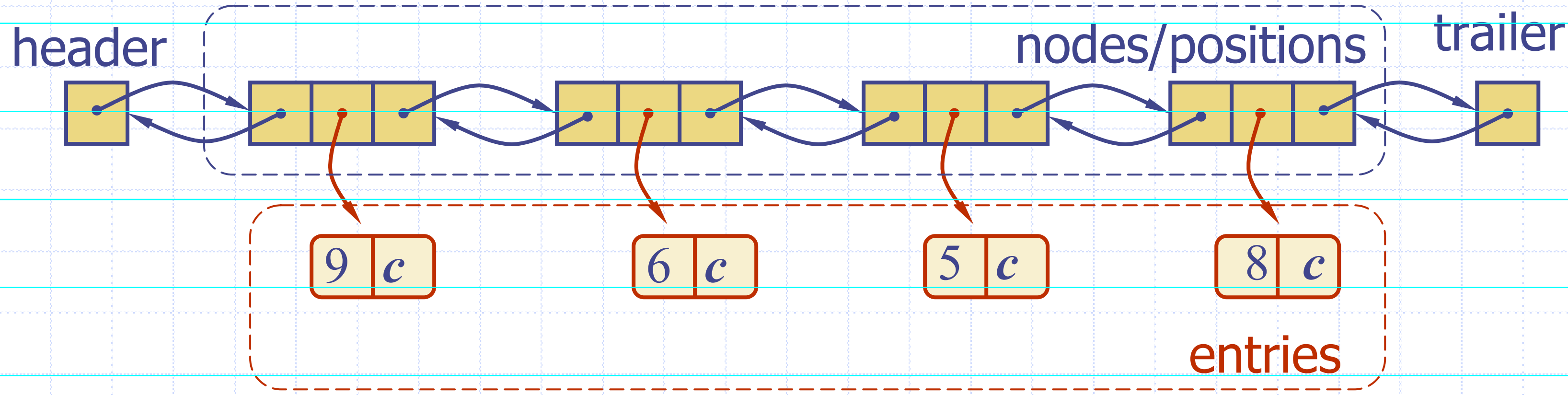
Operation	*Output*	*Map*
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

Comparison to java.util.Map

Map ADT Methods	java.util.Map **Methods**
size()	size()
isEmpty()	isEmpty()
get(*k*)	get(*k*)
put(*k,v*)	put(*k,v*)
remove(*k*)	remove(*k*)
keys()	keySet().iterator()
values()	values().iterator()

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The get(k) Algorithm

Algorithm get(*k*):
  B = *S*.positions() {*B* is an iterator of the positions in *S*}
  while *B*.hasNext() **do**
    p = *B*.next() {the next position in *B*}
    if *p*.element().key() = *k* **then**
      return *p*.element().value()
  return null {there is no entry with key equal to *k*}

The put(k,v) Algorithm

Algorithm put(k,v):
B = S.positions()
while B.hasNext() **do**
    p = B.next()
    if p.element().key() = k **then**
        t = p.element().value()
        B.replace(p,(k,v))
        return t {return the old value}
S.insertLast((k,v))
n = n + 1 {increment variable storing number of entries}
return null {there was no previous entry with key equal to k}

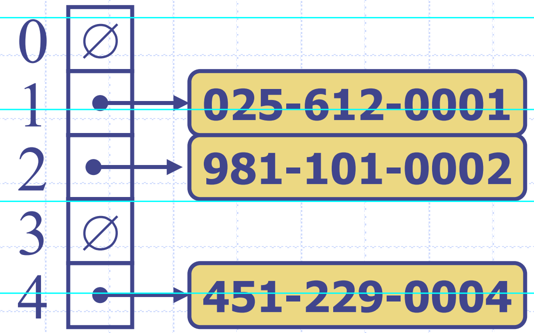
The remove(k) Algorithm

Algorithm remove(k):
$B = S$.positions()
while B.hasNext() **do**
    $p = B$.next()
    if p.element().key() = k **then**
        $t = p$.element().value()
        S.remove(p)
        $n = n - 1$ {decrement number of entries}
        return t {return the removed value}
return null   {there is no entry with key equal to k}

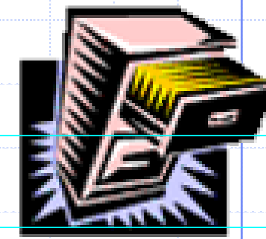
Performance of a List-Based Map

- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Tables



Recall the Map ADT (§ 8.1)



- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoicated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M

U is universe of keys.
S is evolving set of keys

[1] Array based: $O(1)$ operations but $O(|U|)$ space
[2] list based: $O(|S|)$ space but $O(|S|)$ lookup
Desire: $h: U \rightarrow \{0,1,2,\ldots n-1\}$

Q: What to do with collisions

Ignore leap year:

THE BIRTHDAY PARADOX

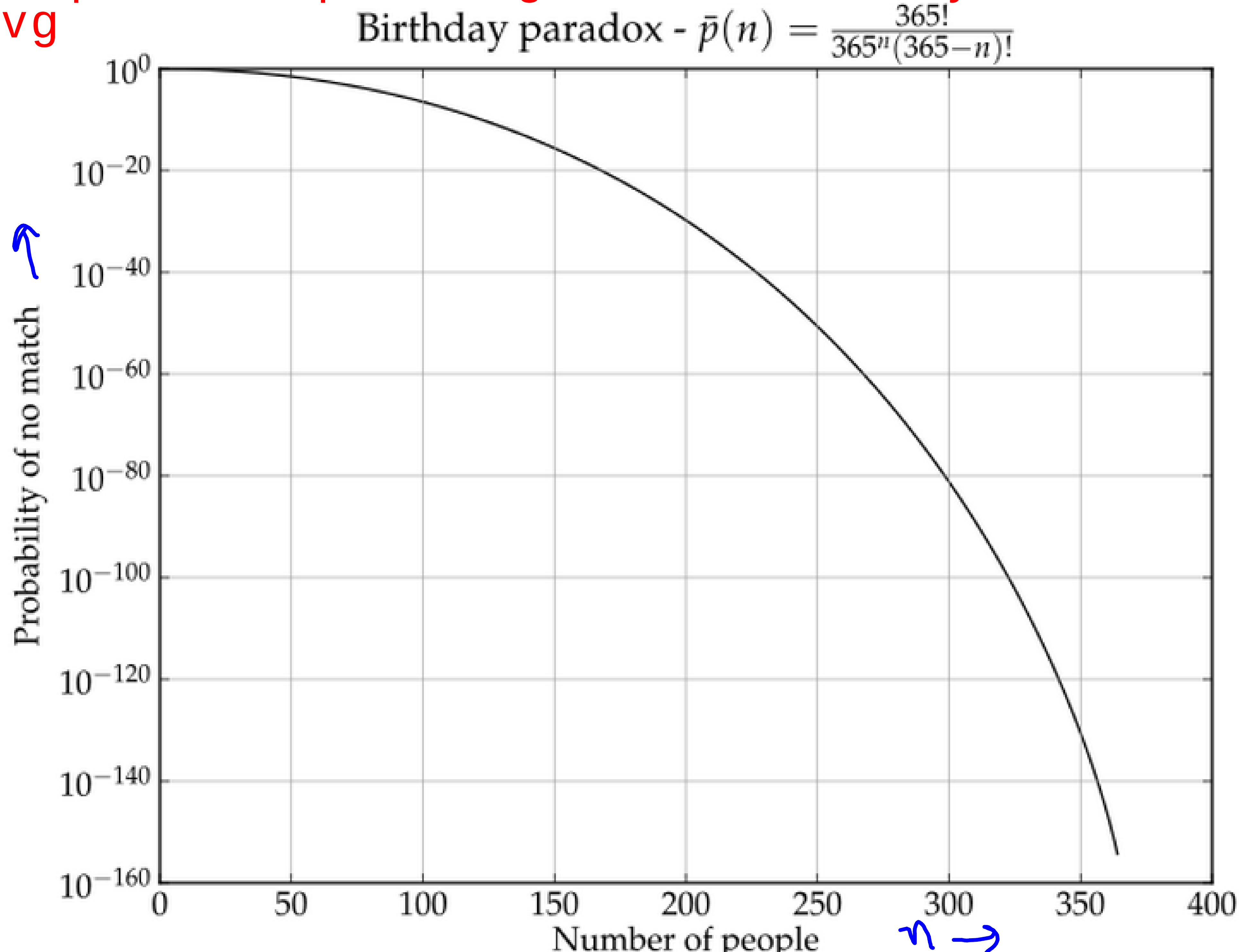
Consider n people with random birthdays (i.e. with each day of the year equally likely). How large does n need to be before there is atleast a 50% chance that two people have the same birthday?

Choose: $\underbrace{23}_{50\%}$, $\underbrace{57}_{99\%}$, $\underbrace{184}_{99.99\%}$, $\underbrace{367}_{100\%}$

P(amongst n people atleast 2 have same bday)

$$= 1 - P(\text{all } n \text{ people have diff bdays}) = -\frac{{}^{365}P_n}{(365)^n} + 1$$

http://en.wikipedia.org/wiki/File:Birthdaymatch.svg



# of people (n) = # of possible keys

# of days in a year (365) = # of slots that keys will be mapped to using hash function

Deterministic

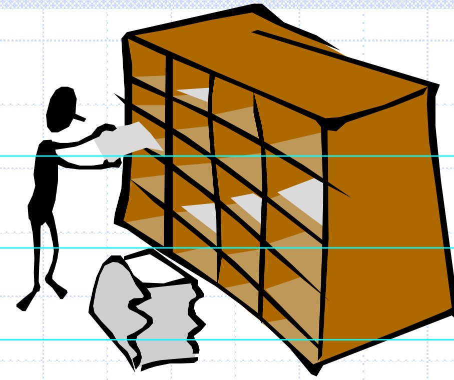
Good hash fn ⟷ Strategy for collision handling

both reqd

↓ Best (non-practical) hash function (in terms of minimum collisions) is RANDOM

↓ In practice we need deterministic hash fns. that closely mimic random

Hash Functions and Hash Tables (§ 8.2)



- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$ If $x = 201$ & $N = 10$ $h(x) = 1$
- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of : $U \to [0, N-1]$ Not: $O(|U|)$
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

(1) If x (keys) are distributed uniformly over all integers, then choice of N does not affect distribution. You will only choose N based on how much space/ resources are available

(2) If x (keys) are not all equally possible (for eg: names of people have some patterns in a region) then $x \bmod N$ could lead to unequal slot occupancy. Eg: x only evens & $N=10$

$\therefore$ In such a case, the "uniform" hashing is ideal & realised, for eg by $\boxed{N = \text{large prime}}$

x = all even

$N = 10 \Rightarrow x \bmod N \in \{0, 2, 4, 6, 8\}$

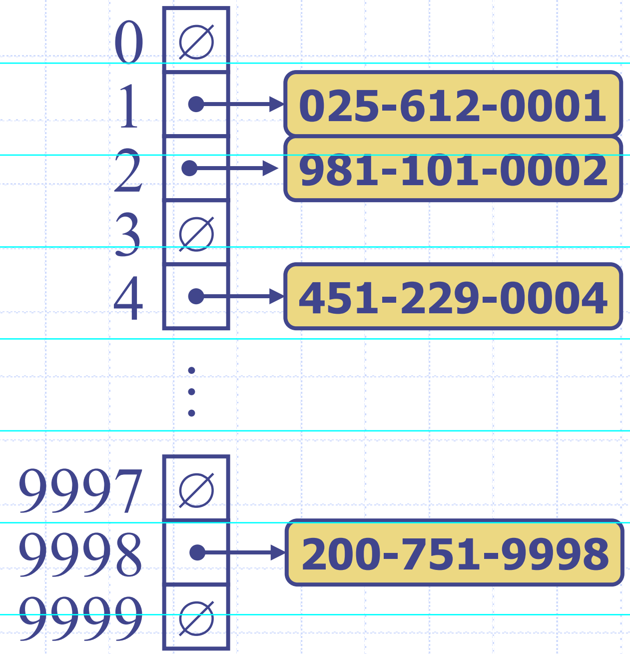
$N = 11 \Rightarrow x \bmod N \in \{0, 1, 2, \ldots 9, 10\}$

$N = 2 \Rightarrow x \bmod N \in \{0, 1\}$

Primes are good. To reduce collisions, large prime prefferred

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) =$ last four digits of x



Application:
1] IP address lookup at a router requires hash table..
2] The 2-sum problem (repeated lookups) in constant time!
* In general, for search algorithms: search tree algorithms, searching on graphs, chess playing program (search based configuration to find next possible moves)

} H/w similar to histogram

3] Lookups for compilers! Symbol tables!
4] Blocking network traffic

Hash Functions (§ 8.2.2)

- A hash function is usually specified as the composition of two functions: Ex: Think of Possibilities

Neither needs to be one-one

Hash code:

h_1: keys $\rightarrow$ integers

Compression function:

h_2: integers $\rightarrow [0, N-1]$

$h_2 = x \bmod N$

where N is a large prime is frequent choice

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way

Example hash codes:

1. Binary representation of the "key" object
   Eg: "serialised object" in java.
2. The "binary" address of the object
3. For strings: Sum of ASCII values of characters.
4. For strings: Since "character sequences" are often characteristic in strings (eg: "qz" is unlikely to occur), position dependent encoding of characters

Eg: $$\sum_{i=1}^{|s|} \text{Ascii}(c_i) * N_1^{\,i-1}$$

where $S = c_1 c_2 \ldots c_{|s|}$

(imagine writing the string with $N_1 = 26$ in base 26)

Eg: $N_1 = 26$

$$h_1 : S \xrightarrow[N_1]{} x$$

$$h_2 : x \xrightarrow[x \bmod N_2]{} j \in [0, N_2]$$

Q: will some relation between N_1 & N_2 help in practice? ANS: N_1 & N_2 coprime

Eg: $N_2 = 8$ & x = multiples of $3 \Rightarrow$ all 8 slots filled up!

Hash Codes (§ 8.2.3)

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0\, a_1\, \ldots\, a_{n-1}$$

  - We evaluate the polynomial

    z^i for offset

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$

    at a fixed value z, ignoring overflows

    Optimise multiplications

  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

    experimental results. Imp for assignment

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

    $$p_0(z) = a_{n-1}$$

    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$

    $$(i = 1, 2, \ldots, n-1)$$

- We have $p(z) = p_{n-1}(z)$

Since for strings such as names/ids, char positions imp.

Compression Functions (§ 8.2.4)

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

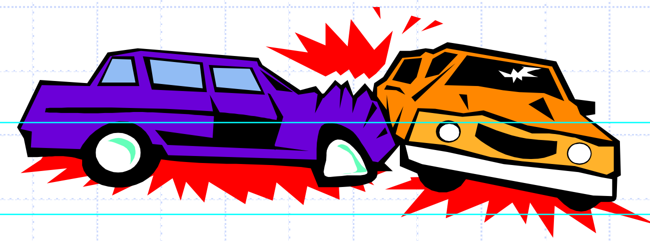
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that

    $a \bmod N \neq 0$

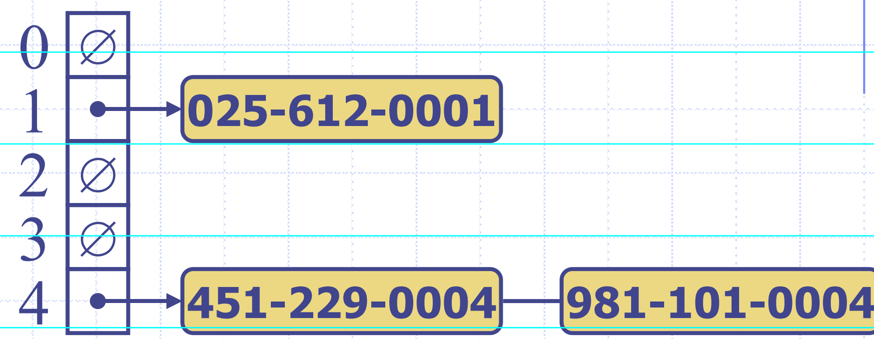
    else y will be meaningless
  - Otherwise, every integer would map to the same value b

If polynomial encoding (accummulation) then prefer: z & N are co prime

Collision Handling (§ 8.2.5)



- Collisions occur when different elements are mapped to the same cell



- **Separate Chaining**: let each cell in the table point to a linked list of entries that map there
- Separate chaining is simple, but requires additional memory outside the table (pointers)

Map Methods with Separate Chaining used for Collisions

- Delegate operations to a list-based map at each cell:

Algorithm get(k):
Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map
return $A[h(k)]$.get(k) {delegate the get to the list-based map at $A[h(k)]$}

Invoking one data structure within another

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return **null**
$t = A[h(k)]$.put(k,v) {delegate the put to the list-based map at $A[h(k)]$}
if t = **null then** {k is a new key}
    $n = n + 1$
return t

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map
$t = A[h(k)]$.remove(k) {delegate the remove to the list-based map at $A[h(k)]$}
if $t \neq$ **null then** {k was found}
    $n = n - 1$
return t

Linear Probing

(Somewhat overcomes memory wasted in pointers stored in linked lists)

- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇓

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

1 1 2 1 3 1 5 4

1 1 2 1 1 1 3 1

Exercise: Compare total cost of linear Probing with linked list for this problem