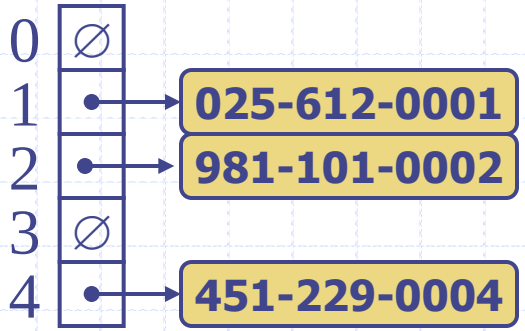
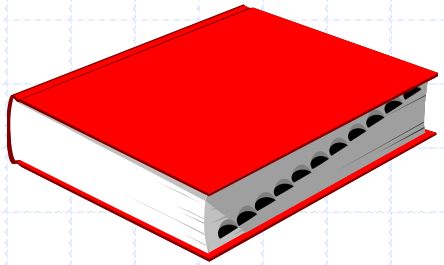
Dictionaries and Hash Tables



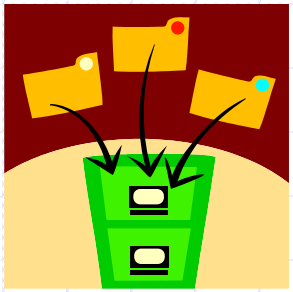
Dictionary ADT



- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)

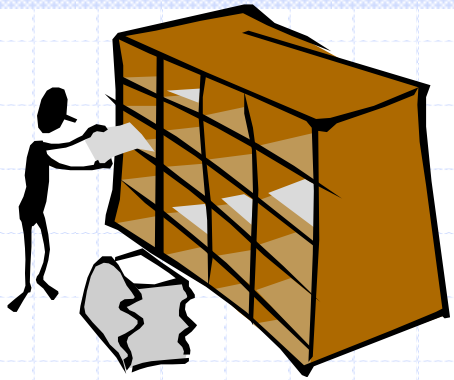
- Dictionary ADT methods:
  - find(k): if the dictionary has an item with key k, returns the position of this element, else, returns a null position.
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element. An error occurs if there is no such element.
  - size(), isEmpty()
  - keys(), Elements()

Log File



- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - find and removeElement 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 log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Functions and Hash Tables



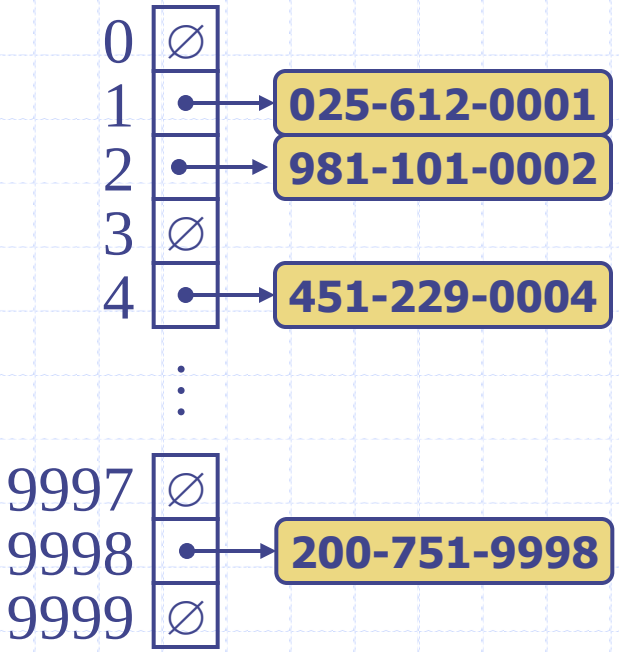
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

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

  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
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a dictionary storing items (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



Hash Functions

- Ideally, storage and access performance is $O(1)$!
- A hash function is usually specified as the composition of two functions:

  Hash code map:

  h_1: keys $\rightarrow$ integers

  Compression map:

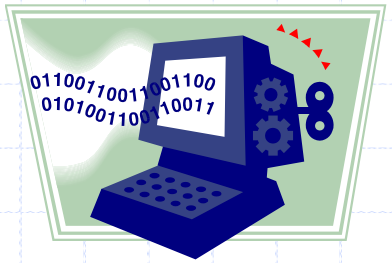
  h_2: integers $\rightarrow [0, N-1]$

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

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

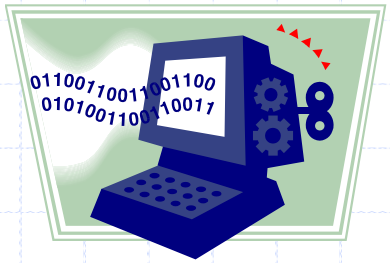
- The goal of the hash function is to "disperse" the keys in an apparently uniform random way -- but it is ***not*** random!

Example Hash Code Maps



- Memory address:
  - We reinterpret the memory address of the key object as an integer
    - e.g.,
      - Data *ptr;
      - int key = (int)ptr;
  - Good, but can do better for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
    - e.g., "J23" -> binary
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., char, short, int and float)
- 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)
    - e.g., "example" maps to 5+24+1+13+16+12+5 =76
  - 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 on many machines)

Example Hash Code Maps



- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    "some text" = $a_0 a_1 \ldots a_{n-1}$
  - We evaluate the polynomial
    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- 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 $p(z) = p_{n-1}(z)$

Example

- Encode "Purdue" using
  - $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$ with z=33

ASCII table

ascii	num	ascii	num	ascii	num
[sp]	32	A	65	a	97
!	33	B	66	b	98
"	34	C	67	c	99
#	35	D	68	d	100
$	36	E	69	e	101
%	37	F	70	f	102
&	38	G	71	g	103
'	39	H	72	h	104
(	40	I	73	i	105
)	41	J	74	j	106
*	42	K	75	k	107
+	43	L	76	l	108
,	44	M	77	m	109
-	45	N	78	n	110
.	46	O	79	o	111
/	47	P	80	p	112
0	48	Q	81	q	113
1	49	R	82	r	114
2	50	S	83	s	115
3	51	T	84	t	116
4	52	U	85	u	117
5	53	V	86	v	118
6	54	W	87	w	119
7	55	X	88	x	120
8	56	Y	89	y	121
9	57	Z	90	z	122
:	58	[	91	{	123
;	59	\	92	|	124
<	60	]	93	}	125
=	61	^	94	~	126
>	62	_	95	[del]	127
?	63	`	96		
@	64				

Alphabet position

A=1
B=2
C=3
D=4
E=5
F=6
G=7
H=8
I=9
J=10
K=11
L=12
M=13
N=14
O=15
P=16
Q=17
R=18
S=19
T=20
U=21
V=22
W=23
X=24
Y=25
Z=26

Example

- Encode "Purdue" using
  - $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$ with z=33
- Using alphabet-position
  - $p(z) = 16+(21*33)+(18*33^2)+(4*33^3)+(21*33^4)+(5*33^5)$
  - $p(z) = 220{,}956{,}235$

Example Compression Maps

- Division:
  - $h_2(y) = y \bmod N$
  - Note: 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
  - However, be careful choosing a and b

Example Compression Maps

- Multiply, Add and Divide (MAD?):
  - $h_2(y) = (ay + b) \bmod N$
- Pick a=7, b=3, N=14
  - $h_2(0) = 3$
  - $h_2(1) = 10$
  - $h_2(2) = 3$
  - $h_2(3) = 10$
  - $h_2(4) = 3$
- Notice a pattern?

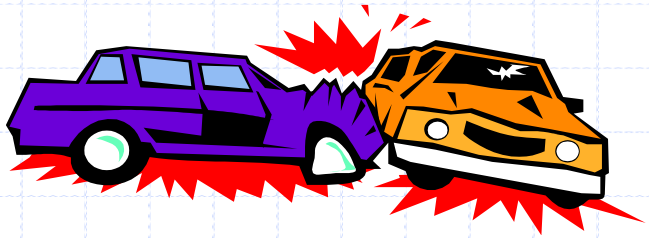
Example Compression Maps

- Multiply, Add and Divide (MAD?):
  - $h_2(y) = (ay + b) \bmod N$
- Pick a and N such that (a mod N) != 0
- a=7, b=3, N=17
  - $h_2(0) = 3$
  - $h_2(1) = 10$
  - $h_2(2) = 0$
  - $h_2(3) = 7$
  - $h_2(4) = 14$
  - Etc…

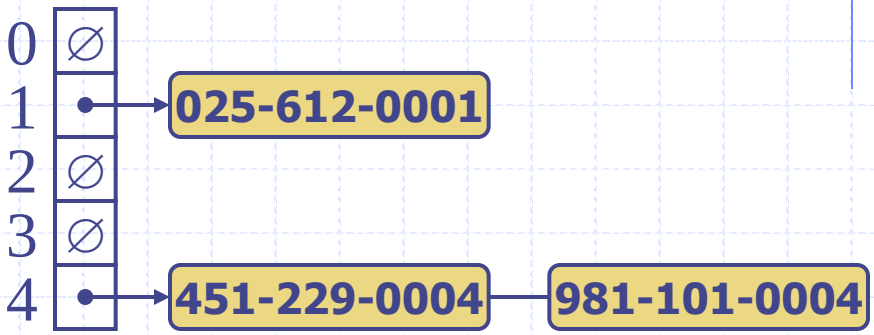
Example Compression Maps

- Multiply, Add and Divide (MAD?):
  - $h_2(y) = (ay + b) \bmod N$
- Pick N as prime
- Pick a and N such that (a mod N) != 0
- Recommendation:
  - Pick a and b as prime numbers
  - e.g.
    - $h_2(y) = (7y + 3) \bmod 13$
    - $h_2(y) = (21y + 17) \bmod 101$

Collision Handling



- Collisions occur when different elements are mapped to same cell
- **Chaining**: each cell points to a linked list of elements
  - Simple but requires additional memory outside the table



- Performance (briefly):
  - InsertItem
    - O(1) on average but can be O(N) in worst-case
  - RemoveItem
    - O(1) on average but can be O(N) in worst-case

Example

- $h(k) = k \bmod 6$

A=1
B=2
C=3
D=4
E=5
F=6
G=7

H=8
I=9
J=10
K=11
L=12
M=13
N=14

O=15
P=16
Q=17
R=18
S=19
T=20
U=21

V=22
W=23
X=24
Y=25
Z=26

Alphabet position

A=1
B=2
C=3
D=4
E=5
F=6
G=7

H=8
I=9
J=10
K=11
L=12
M=13
N=14

O=15
P=16
Q=17
R=18
S=19
T=20
U=21

V=22
W=23
X=24
Y=25
Z=26

Collision Handling

- So what can we do to keep the all data in the table?
  - e.g., O(N) space
- Probing: try a different space in the table
  - Must be repeatable
  - Must find space fast
  - Must support find/insert/remove

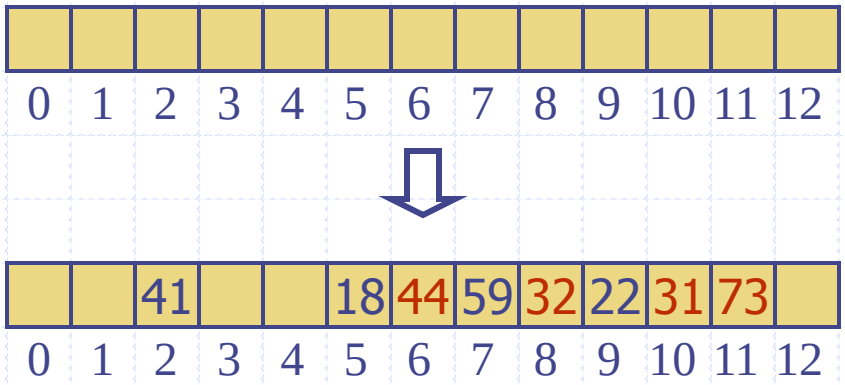
Collision Handling

- Solution: when a collision for key k occurs at i, compute a new index i'
  - If $i=h(k)$ is "occupied"
  - Then, compute $i'=((i+f(j)) \bmod N)$ for attempt j
- Linear probing: e.g. $f(j)=j$
- Quadratic probing: e.g. $f(j)=j^2$
- Double Hashing: e.g. $f(j)=j(q-(k \bmod N))$
  - "compute another hash value"
- Other schemes?

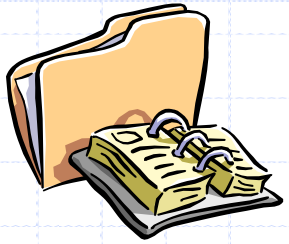
Linear Probing

- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
  - e.g., $i' = (i+1) \bmod N$
- Uses a concept called "open addressing": the colliding item is placed in a different cell of the table
- Each table cell inspected is referred to as a "probe"

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



Search with Linear Probing

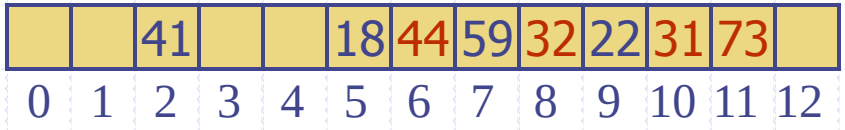


- Consider a hash table A that uses linear probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following three occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm find(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return Position(null)
        else if c.key () = k
            return Position(c)
        else
            i ← (i + 1) mod N
            p ← p + 1
    until   p = N
    return Position(null)
```

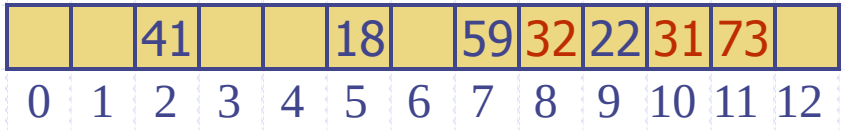
Updates with Linear Probing

- Insertions and deletions are a little more complicated…
- Example:
  - $h(x) = x \bmod 13$
  - Keys 18, 41, 22, 44, 59, 32, 31, 73 were inserted in this order
  - Let's "remove" 44…



Updates with Linear Probing

- Insertions and deletions are a little more complicated…
- So what can we do?
- Welcome to the world of special markers…
- Example:
  - $h(x) = x \bmod 13$
  - Keys 18, 41, 22, 44, 59, 32, 31, 73 were inserted in this order
  - Let's "remove" 44…
  - Let's "search" for 32…
  - Slot 6=32 mod 13 is empty…"not found"
  - Oops!



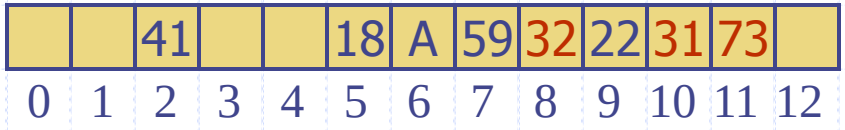
Updates with Linear Probing

- Insertions and deletions are a little more complicated
- We introduce a special item, called $AVAILABLE$, which replaces deleted elements
  - $AVAILABLE$ *!= empty*
  - Why do we need this special item?
- removeElement(k)
  - We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item $AVAILABLE$ and we return the position of this item
  - Else, we return a null position
- insertItem(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores $AVAILABLE$, or
    - N cells have been unsuccessfully probed
  - We store item (k, o) in cell i

Updates with Linear Probing

- Put "A" in slots of removed item
- Compute slot for 32, skip over "A" and search until empty or N cells

- Example:
  - $h(x) = x \bmod 13$
  - Keys 18, 41, 22, 44, 59, 32, 31, 73 were inserted in this order
  - Let's remove 44…
  - Let's search for 32…
  - Found at slot 8…
  - Groovy!



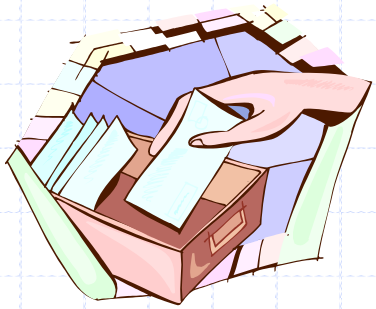
Linear Probing

- Advantage:
  - Relatively simple to implement
    - Though special care must be taken to handle removals from the hash table
- Disadvantage:
  - Upon successive collisions, items tend to "clump together"
- How can we overcome the above disadvantange?

Quadratic Probing

- Handles collisions by placing the colliding item in $i' = (i+j^2) \bmod N$ for $j=0,1,2,\ldots$
- Similar find/insert/remove operations as with linear probing
- Collided items tend to be more distributed
- However:
  - It might take "lots of bouncing around the table" to find an item (or empty slot)

Double Hashing



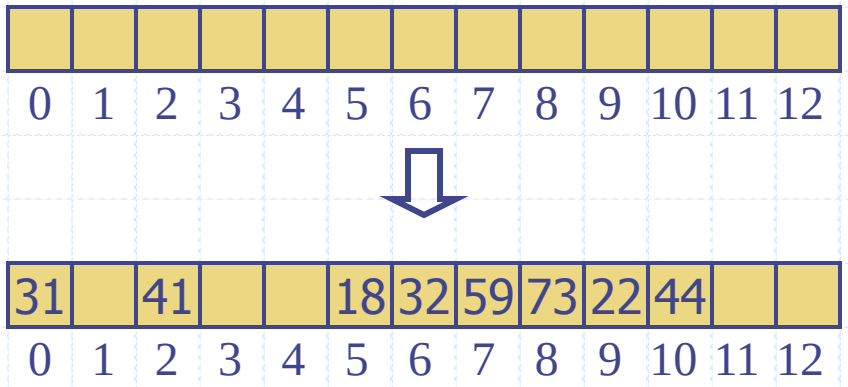
- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(i + jd(k)) \bmod N$$
for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression map for the secondary hash function:
$$d_2(k) = q - (k \bmod q)$$
where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
$$1, 2, \ldots, q$$

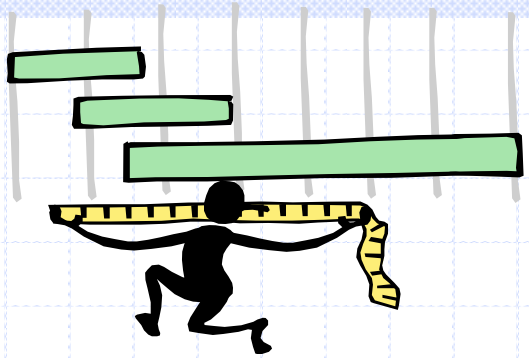
Example of Double Hashing

- 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		

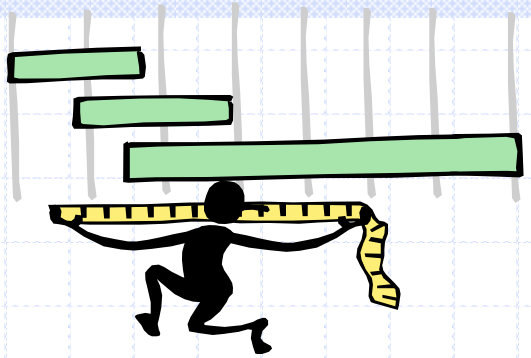


Performance of Hashing



- 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 dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, typical insertion time is $\alpha * 0.5$

Performance of Hashing



- 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 dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, typical insertion time is $\alpha * 0.5$
- 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 large
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

End

Universal Hashing

- A family of hash functions is **universal** if, for any $0 \le i,j \le M-1$,
  $\Pr(h(j)=h(k)) \le 1/N$.
- Choose p as a prime between M and 2M.
- Randomly select $0<a<p$ and $0 \le b<p$, and define $h(k)=(ak+b \bmod p) \bmod N$

- Theorem: The set of all functions, h, as defined here, is universal.

Proof of Universality (Part 1)

- Let f(k) = ak+b mod p
- Let g(k) = k mod N
- So h(k) = g(f(k)).
- f causes no collisions:
  - Let f(k) = f(j).
  - Suppose k<j. Then

$$aj + b - \left\lfloor \frac{aj + b}{p} \right\rfloor p = ak + b - \left\lfloor \frac{ak + b}{p} \right\rfloor p$$

$$a(j - k) = \left(\left\lfloor \frac{aj + b}{p} \right\rfloor - \left\lfloor \frac{ak + b}{p} \right\rfloor \right) p$$

- So a(j-k) is a multiple of p
- But both are less than p
- So a(j-k) = 0. i.e., j=k. (contradiction)
- Thus, f causes no collisions.

Proof of Universality (Part 2)

- If f causes no collisions, only g can make h cause collisions.
- Fix a number x. Of the p integers y=f(k), different from x, the number such that g(y)=g(x) is at most $\lceil p / N \rceil - 1$
- Since there are p choices for x, the number of h's that will cause a collision between j and k is at most

$$p\left(\lceil p / N \rceil - 1\right) \leq \frac{p(p-1)}{N}$$

- There are p(p-1) functions h. So probability of collision is at most

$$\frac{p(p-1)/N}{p(p-1)} = \frac{1}{N}$$

- Therefore, the set of possible h functions is universal.