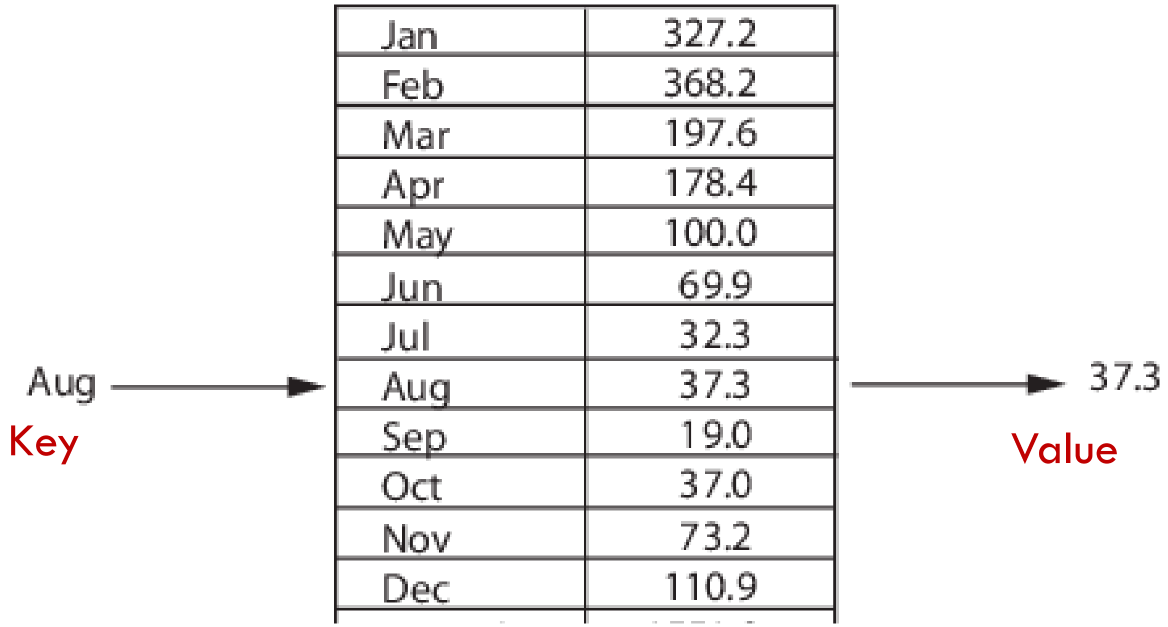
IT309: Maps and Hashtables

IT309: Maps

MAPS OR DICTIONARIES IN PYTHON(1/2)

- A **map** is a searchable collection of items that are key-value pairs
- The main operations of a map are searching, inserting, and deleting items
- Multiple items with the same key are not allowed
- There is no indexing mechanism in maps (unlike linear arrays, lists).
- The Access Time to a data element is in fact O(1) on by using a unique key.

An entry or item : <key,value>



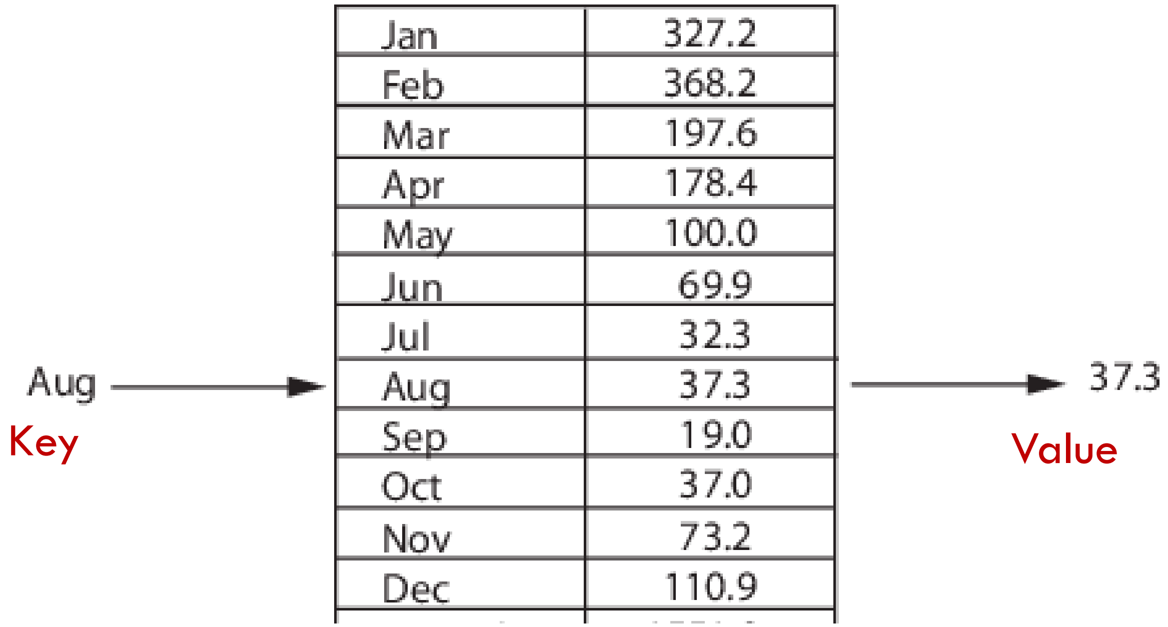
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9

https://www.mathworks.com/help/matlab/matlab_prog/overview-of-the-map-data-structure.html?requestedDomain=www.mathworks.com

MAPS OR DICTIONARIES IN PYTHON (2/2)

- Python's **dict** class represents an abstraction known as a dictionary.
- A dictionary contains unique keys which are mapped to associated values.
- Dictionaries are also known as **associative arrays** or **maps.**

An entry or item : <key,value>



Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9

https://www.mathworks.com/help/matlab/matlab_prog/overview-of-the-map-data-structure.html?requestedDomain=www.mathworks.com

APPLICATIONS OF MAPS

- A university’s information system, using student id (e.g., g-number) as keys mapped to the student’s associated information.
- The domain-name system (DNS), using host name (like www.wiley.com) as keys mapped to the associated IP address (like 208.215.179.146) .
- Many systems employee a username as a key that maps to that user’s associated profile information.
- A computer’s graphic system using color names (like ‘turquoise’) to map to RGB color codes (like 64,224,208).
- Python uses a dictionary to represent each namespace, mapping an identifying strings, such as ‘pi’ to an associated object such as 3.14159.
- Lots of other applications use the idea of a key mapping to an associated value – information in a database, for example

THE FIVE CORE FUNCTIONS OF A MAP ‘M’

Map ADT

1. --getitem-- : M[k] returns the value v associated with key k in map, if one exits, otherwise, raise a KeyError.
2. --setitem-- : Associate value v with key k in map M (M[k] = v), replacing the existing value if the map already contains an item with the key equal to k.
3. --delitem-- : Remove from map M the item with key equal to k (del M[k]) ; if M has no such item, then raise a KeyError.
4. --len-- : returns the number of items in map M (len(M)).
5. --iter--: the default iteration for a map generates a sequence of keys in the map (iter(M)). This method allows loops of the form *for k in M*.

OTHER METHODS OF MAP M

Map ADT, cont.

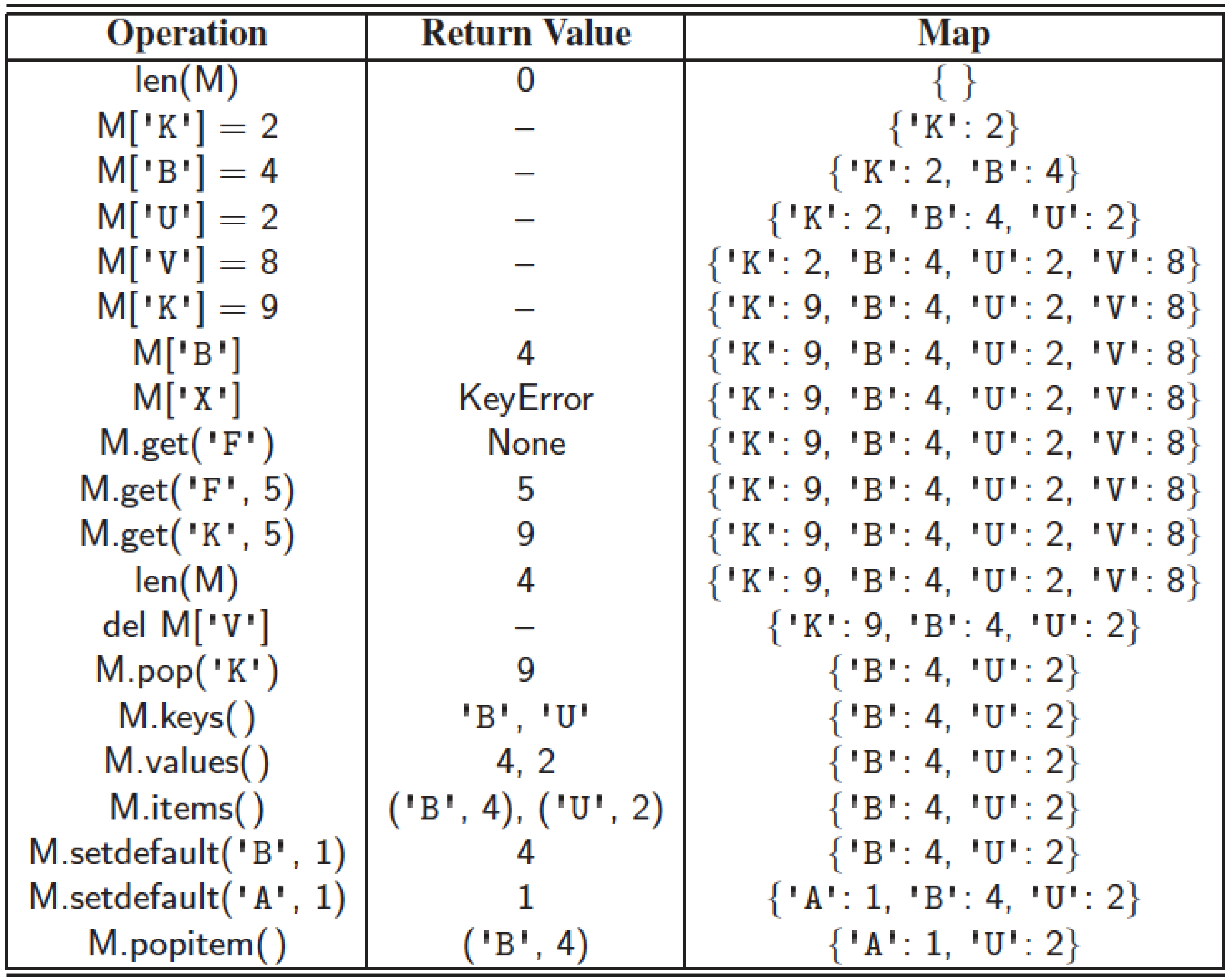
- --contains-- : returns True if the map contains an item with key k.
- get : returns M[k] if key k exists in the map; otherwise returns the default value d. This provides a way of querying the map without the risk of a KeyError.
- setdefault : if key k exists in the map, return M[k]; otherwise, set M[k]=d and return that value (M.setdefault(k,d)).
- pop : remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d , or raise KeyError is parameter d is None (M.pop(k,d=None)).
- popitem: removes an arbitrary key-value pair from the map , and returns a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError (M.popitem()).
- clear : remove all key-value pairs from the map (M.clear()).

OTHER METHODS OF MAP M: CONT.

Map ADT, cont.

- keys : returns a set-like view of all keys of M (M.keys()).
- values : returns a set-like view of all values of M (M.values()).
- items : returns a set-like view of all items of M (M.items()).
- update: Assign M[k] = v for every (k,v) pair in map M2 (M.update(M2)).
- M==M2: return True if maps M and M2 have identical key-value associations.
- M!=M2: return True of maps M and M2 do not have identical ley-value associations.

EXAMPLE



Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys( )	'B', 'U'	{'B': 4, 'U': 2}
M.values( )	4, 2	{'B': 4, 'U': 2}
M.items( )	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem( )	('B', 4)	{'A': 1, 'U': 2}

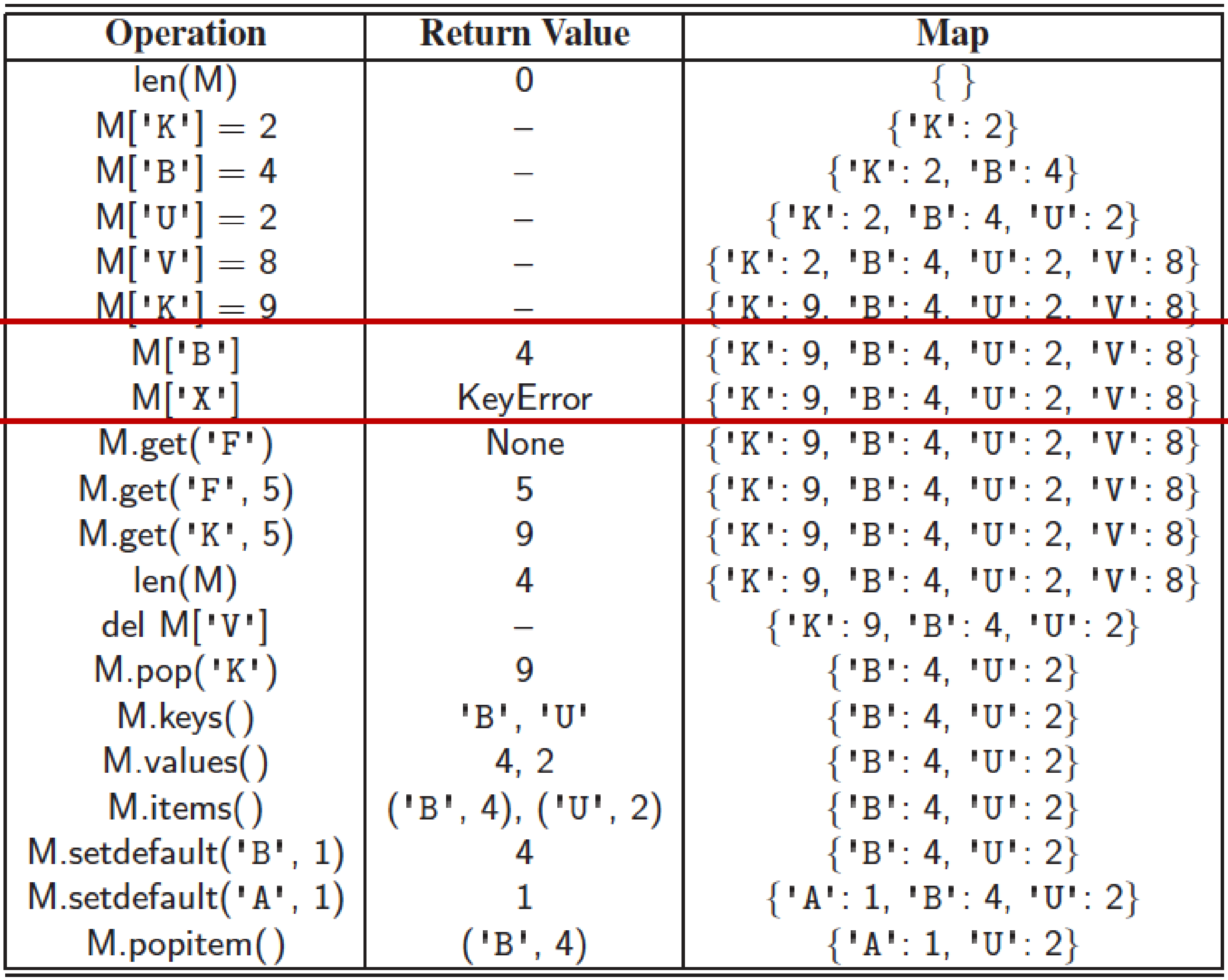
EXAMPLE

Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

EXAMPLE

Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

EXAMPLE



Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

EXAMPLE

Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

APPLICATION: COUNTING WORD FREQUENCIES

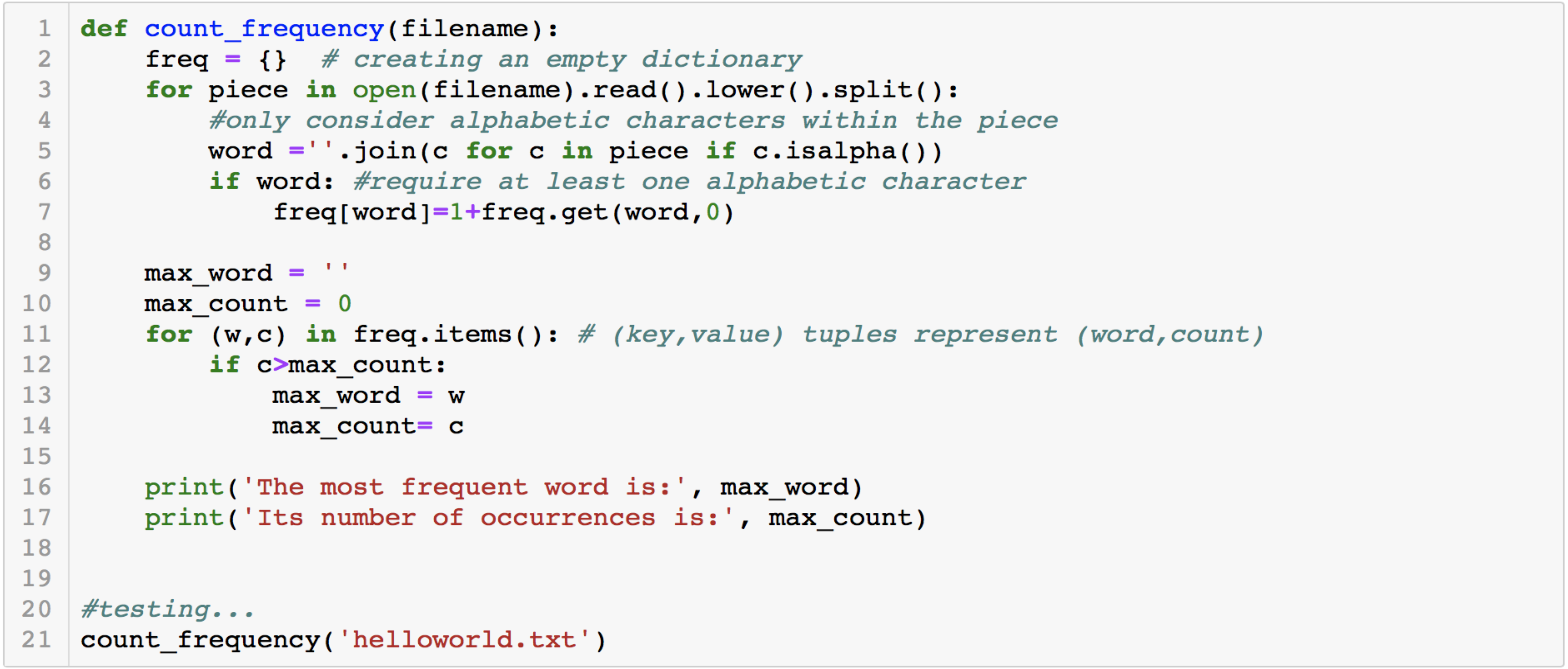
Problem: Counting the number of occurrences of words in a document for statistical analysis using a map data structure.

Solution: consider that each word in the document represents a key k. For each key k, the number of its occurrences in an input document represents its corresponding value v.

Lets use a python dictionary called 'freq':

```
freq[word] = 1 + freq.get(word,0)
```

COUNTING WORD FREQUENCIES: CODING



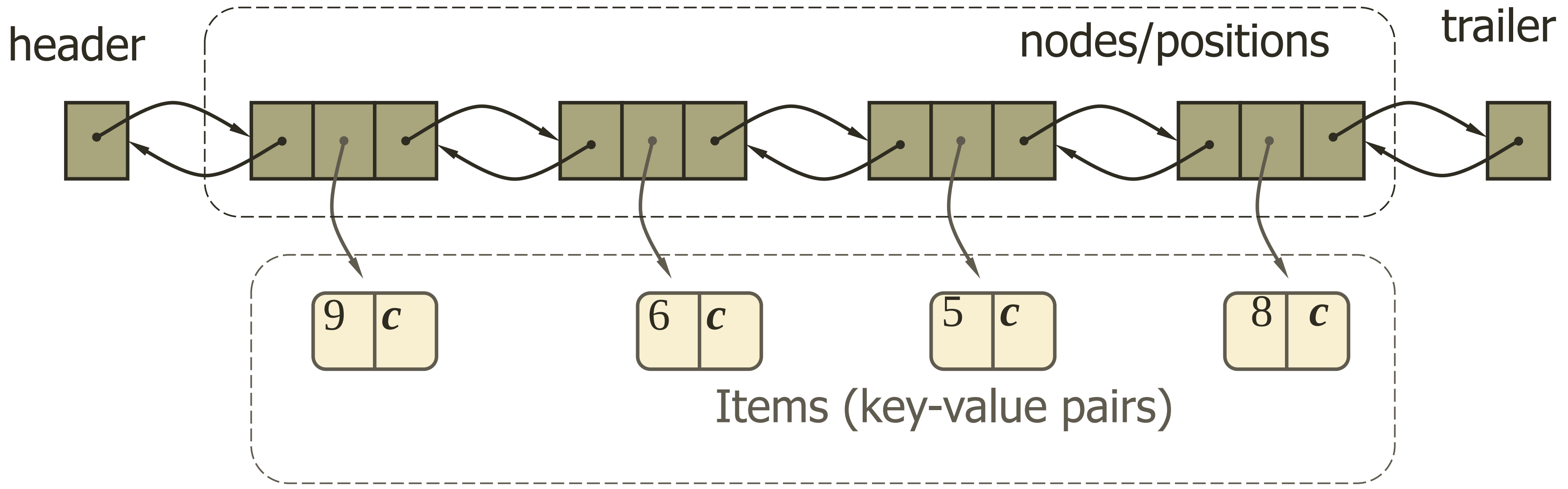
```python
def count_frequency(filename):
    freq = {}  # creating an empty dictionary
    for piece in open(filename).read().lower().split():
        #only consider alphabetic characters within the piece
        word =''.join(c for c in piece if c.isalpha())
        if word: #require at least one alphabetic character
            freq[word]=1+freq.get(word,0)

    max_word = ''
    max_count = 0
    for (w,c) in freq.items(): # (key,value) tuples represent (word,count)
        if c>max_count:
            max_word = w
            max_count= c

    print('The most frequent word is:', max_word)
    print('Its number of occurrences is:', max_count)


#testing...
count_frequency('helloworld.txt')
```

```
The most frequent word is: hello
Its number of occurrences is: 3
```

A SIMPLE LIST-BASED MAP IMPLEMENTATION

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



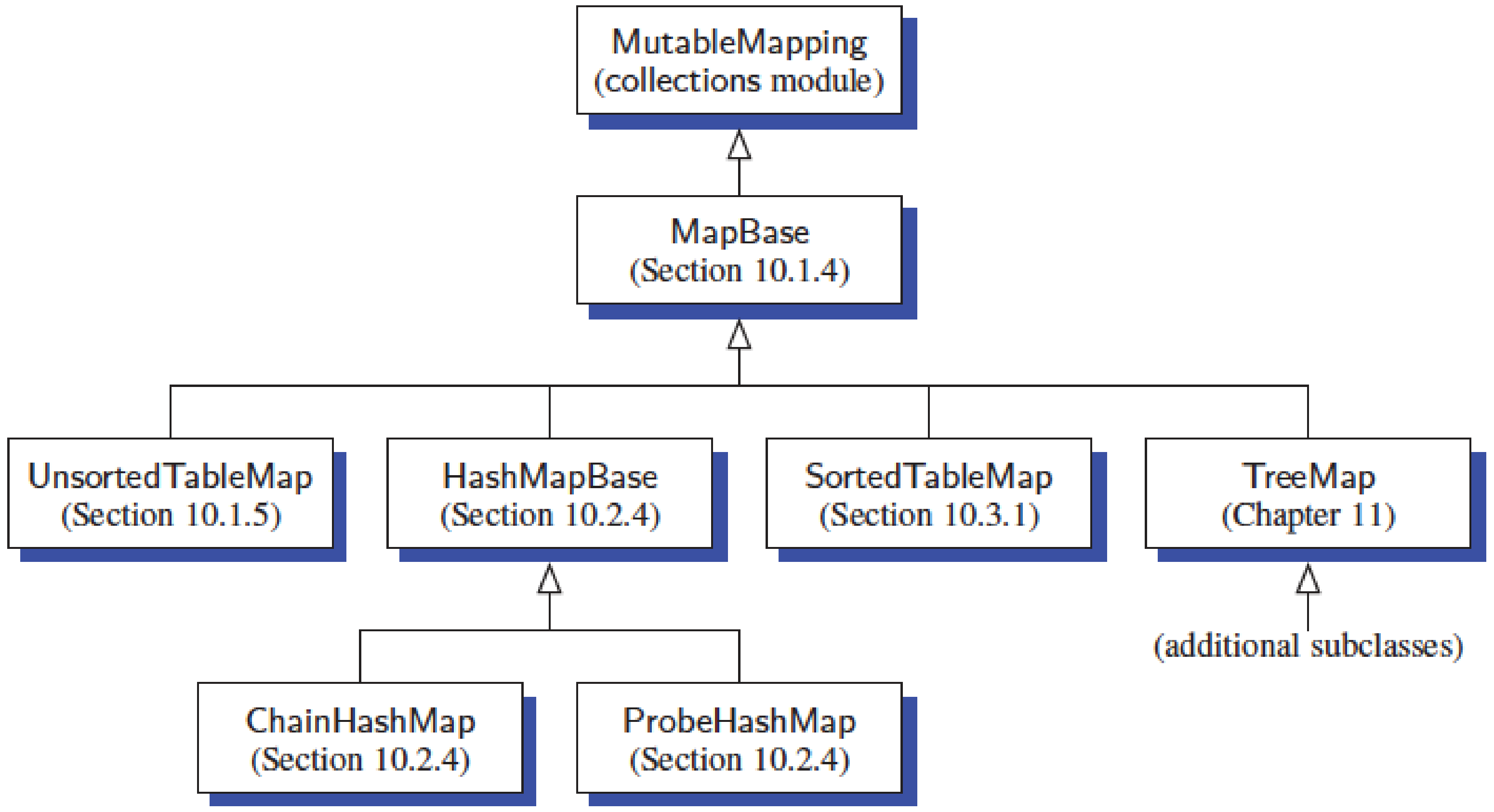
PERFORMANCE OF A LIST-BASED MAP

Performance:

- Inserting an item takes $O(1)$ time since we can insert the new item at the beginning or at the end of the unsorted list
- Searching for or removing an item takes $O(n)$ time, since in the worst case (the item is not found) we traverse the entire list 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 insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

TEXTBOOK IMPLEMENTATION OF MAPS (CHAPTER 10)



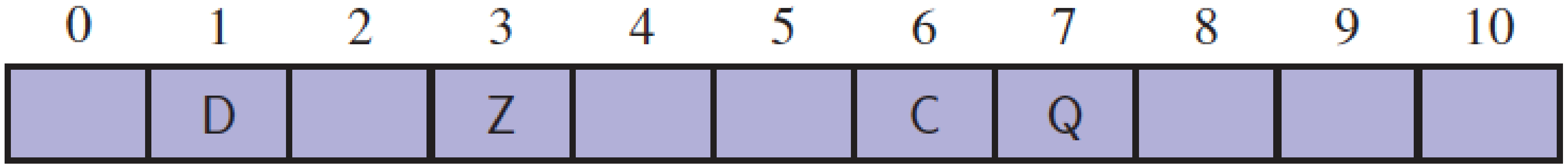
End of Map Overview

Please proceed to the next video

IT309: Hash Functions

RECALL THE NOTION OF A MAP

Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].



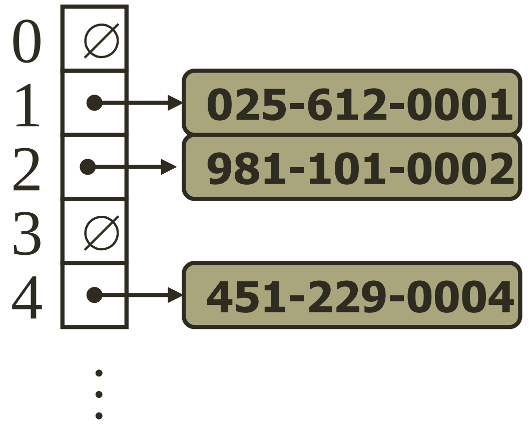
As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to N − 1, for some N ≥ n.

MORE GENERAL KINDS OF KEYS

Q: But what should we do if our keys are not integers in the range from 0 to N – 1?

A: Use a **hash function** to map general keys to corresponding indices in a table.

For instance, the last four digits of a Social Security number.



AN EXAMPLE

- We want to store items that are small nonnegative integers, ranging from 0 to 65,535
- Use an array ‘a’, index 0 to 65,535, initialized to all 0s
- When inserting the item x, execute a[i] = x, where i = the value of the integer
- To find, ensure a[i] ≠ 0, to remove execute a[i] = 0
- All operations are constant time (very desirable)

AN EXAMPLE: TWO PROBLEMS

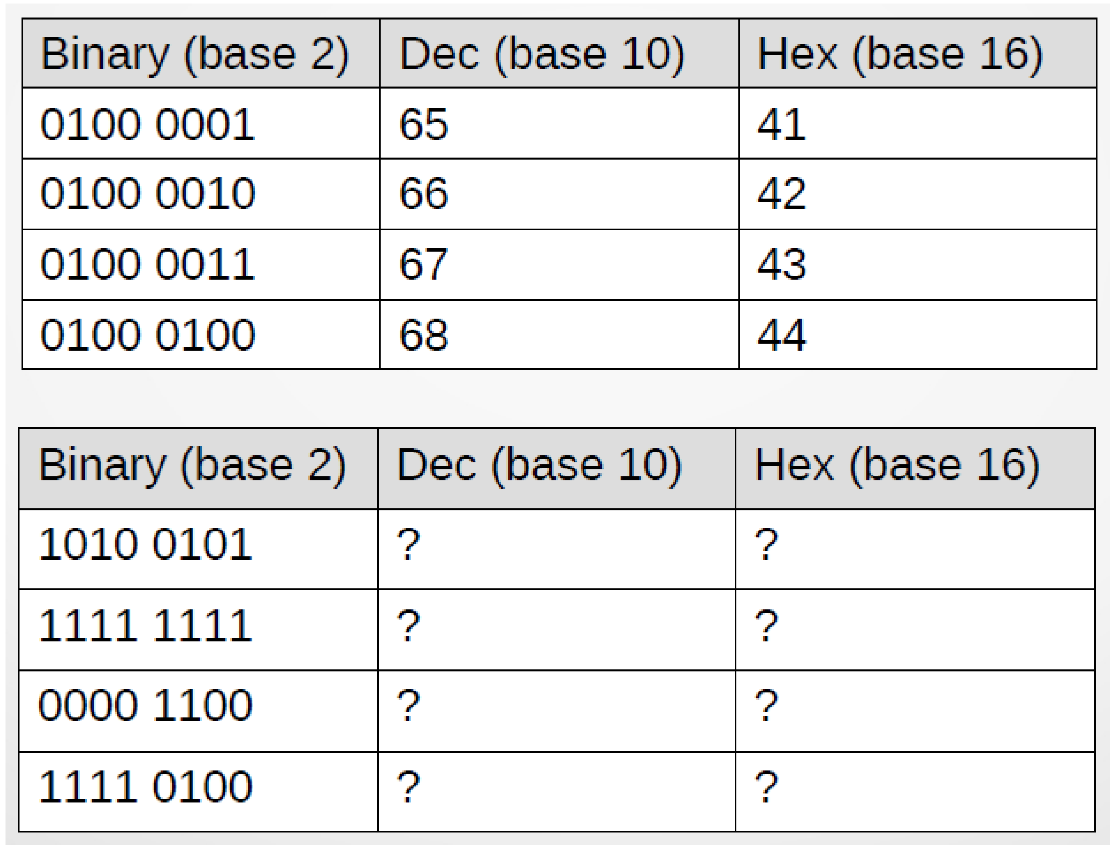
- Problem 1: What do we need to do to store integers greater than 65,535? Storing 32 bit integers, for example, would require an array size of $2^{32} \approx 4.2$ billion elements

- Problem 2: What if we want to store not just integers, but character strings, or other more general data?

QUICK DETOUR: BINARY/HEXADECIMAL NOTATION

HTTP://WWW.ASCIITABLE.COM/

- Converting between binary, decimal, and hexadecimal



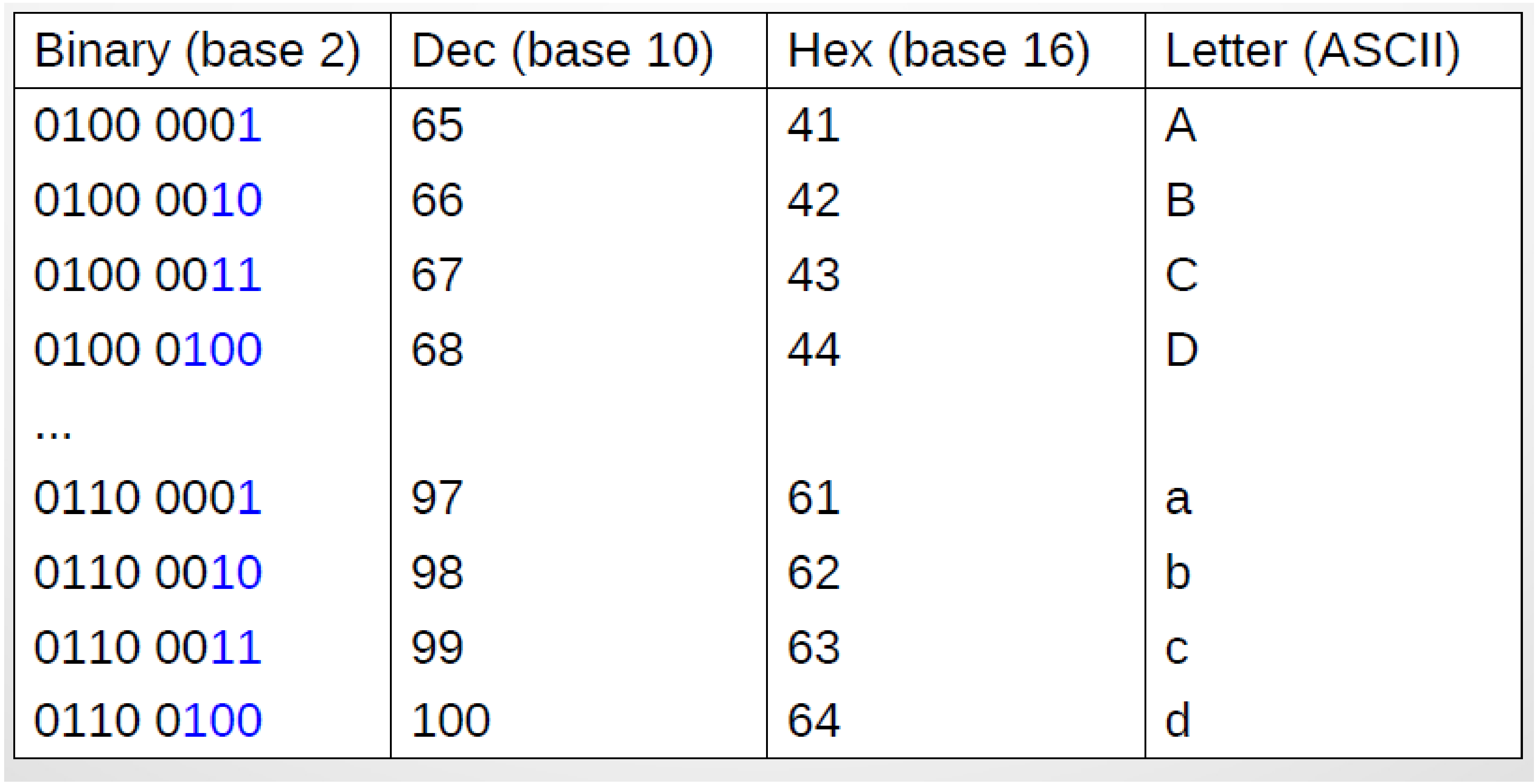
Binary (base 2)	Dec (base 10)	Hex (base 16)
0100 0001	65	41
0100 0010	66	42
0100 0011	67	43
0100 0100	68	44

Binary (base 2)	Dec (base 10)	Hex (base 16)
1010 0101	?	?
1111 1111	?	?
0000 1100	?	?
1111 0100	?	?

INTERNAL REPRESENTATION: NUMBERS AND CHARACTERS

HTTP://WWW.ASCIITABLE.COM/

- All information is represented internally by a string of 0s and 1s

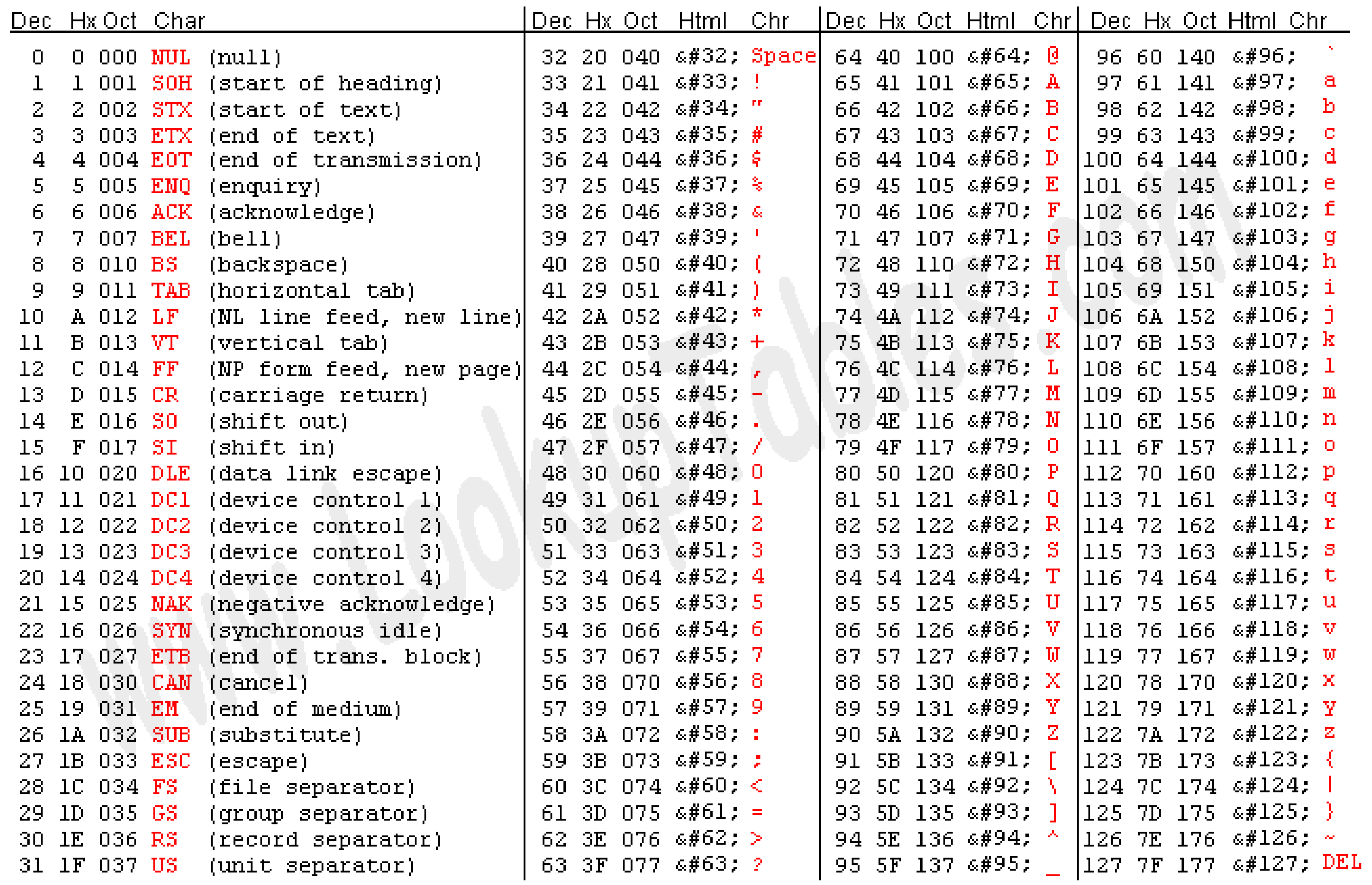


Binary (base 2)	Dec (base 10)	Hex (base 16)	Letter (ASCII)
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
...			
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d

UTF/ASCII CODING

HTTP://WWW.ASCIITABLE.COM/

- Letters are encoded using some sequence of bits. ASCII is subsumed by UTF.



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

]

PROBLEM 2 APPROACH

- Encode character strings using their ASCII equivalents
- Using 7 bit ASCII, "INFS" would be encoded as
  I = 'I' $* 128^3$ = $73 * 128^3$
  N = 'N' $* 128^2$ = $78 * 128^2$
  F = 'F' $* 128^1$ = $70 * 128^1$
  S = 'S' $* 128^0$ = $83 * 128^0$
- ….whose addition (or most any other arithmetic combination) results in a really big number, and would require a very large array
- Problem <u>not</u> solved

GENERALIZED STRING ENCODING

- Example: “cats”
  - Remember, strings are character arrays… and characters are numbers…
    - c = 99, a = 97, t = 116, s = 115
  - $a_0X^{n-1} + a_1X^{n-2} + a_2X^{n-3} + \ldots + a_{n-1}X^0$
    - X = 128 (arbitrary), n = length of array/string

- Computation:
  - $s[0]*128^{n-1} + s[1]*128^{n-2} + s[2]*128^{n-3} + \ldots + s[n-1]*_1 128^0$
  - $99*(128^3) + 97*(128^2) + 116*(128^1) + 115*(128^0)$
  - 209,222,259 = hashcode

SOLVING BOTH PROBLEMS – HASH FUNCTIONS

- Create a function that maps large numbers (or character strings converted into numbers) into smaller, more manageable numbers
- A function that maps an item into a small index is a **hash function**
- If x is an arbitrary positive integer, and tableSize the capacity of our storage array, x%tableSize generates a number between 0 and tableSize – 1 that can be used as an index into the array to store x
- For the "INFS" example, convert the ASCII string into a number, then use the remainder, after dividing by the table size, as the array index

THE GENERAL HASHING PROBLEM

- Variable length input
  - e.g. a string encoded as a number
- Fixed length output
  - e.g. the valid integer indexes of an array
- Problems
  - how do I “abstract” the input so I can store and find it in the given (limited) output space?
  - how do I compute the values?
  - “how can I stuff 10 liters of input into a one liter container?”
  - what other issues occur if I have more values than places to put them?

HASH FUNCTION TYPES - ONE IDEA….USE MODULO DIVISION

- Modulo gives you the remainder after division by some number
  - 3 % 2 = 1
  - 3 % 100 = 3
  - 3 % 3 = 0
  - 6 % 3 = 0
  - 100 % 6 = 4
- This is a common hash function, dividing by certain prime numbers works well

HASH FUNCTION TYPES - ONE IDEA….USE MODULO DIVISION

- Mapping the ASCII version of "INFS" into an array of 100,000 means mapping a number of whose range is ~[0 ….220,000,000] into 100,000 spaces
- Unless we're going to map only a few items into the hash table, it's likely that different inputs will be hash-mapped into the same array space – this is known as a **collision**
- Major issue with using modulo division, and hash functions in general is collisions
- How to handle collisions is a major topic

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
- Storage called a 'table', often implemented with an array, of size N

When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

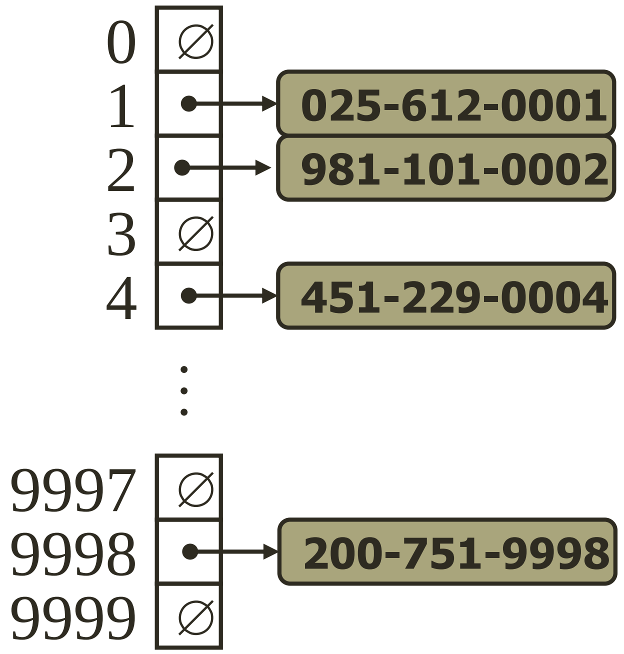
HASHING TERMINOLOGY

- **Hash Function**
  - computes a hash code from an object
- **Hash Code**
  - the integer computed by a hash function
- **Hash Table**
  - Mapping whose keys are hash codes and values are the thing being hashed
  - Look a lot like arrays / linked lists
- **Load (or Load Factor)** = item count / table size (λ)

SSN 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



HASH FUNCTIONS

A hash function can be specified as the composition of two functions:

Hash code:
$\boldsymbol{h}_1$: keys $\rightarrow$ integers

Compression function:
$\boldsymbol{h}_2$: integers $\rightarrow [0, \boldsymbol{N} - 1]$

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

$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$$

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

HASH FUNCTIONS – TWO STEPS

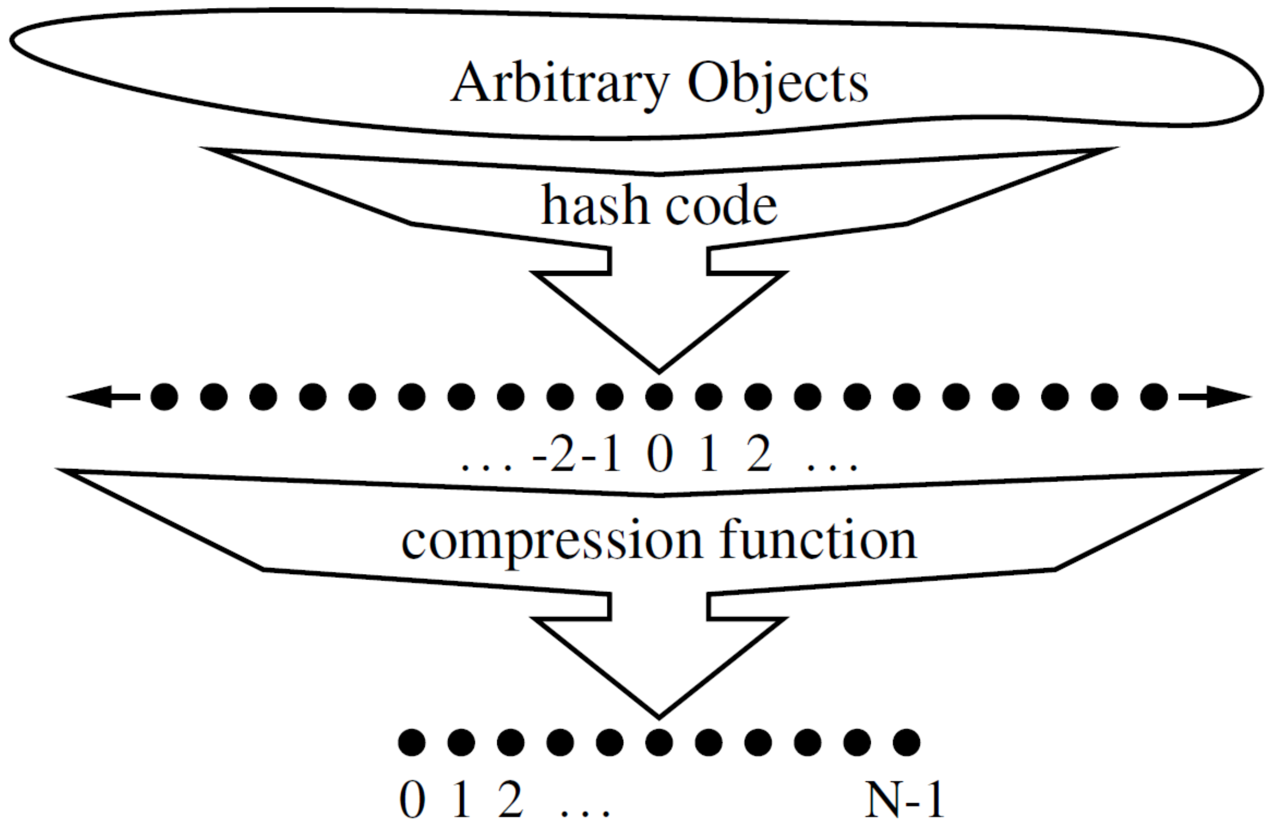
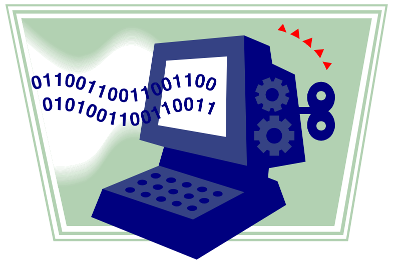


Figure 10.5: Two parts of a hash function: a hash code and a compression function.

HASH CODE VARIETIES



Memory address:

- We reinterpret the memory address of the key object as an integer - good in general, can be tricky for 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 highest value integer we will use

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

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

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

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

COMPRESSION FUNCTIONS

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
- Very common choice

Multiply, Add and Divide (MAD):

- $h_2(y) = (ay + b) \bmod N$
- a and b are nonnegative integers such that

$$a \bmod N \neq 0$$

- Otherwise, every integer would map to the same value b

ABSTRACT HASH TABLE CLASS

```
class HashMapBase(MapBase):
  """Abstract base class for map using hash-table with MAD compression."""

  def __init__(self, cap=11, p=109345121):
    """Create an empty hash-table map."""
    self._table = cap * [ None ]
    self._n = 0                             # number of entries in the map
    self._prime = p                         # prime for MAD compression
    self._scale = 1 + randrange(p−1)        # scale from 1 to p-1 for MAD
    self._shift = randrange(p)              # shift from 0 to p-1 for MAD

  def _hash_function(self, k):
    return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

  def __len__(self):
    return self._n

  def __getitem__(self, k):
    j = self._hash_function(k)
    return self._bucket_getitem(j, k)       # may raise KeyError

  def __setitem__(self, k, v):
    j = self._hash_function(k)
    self._bucket_setitem(j, k, v)                  # subroutine maintains self._n
    if self._n > len(self._table) // 2:            # keep load factor <= 0.5
      self._resize(2 * len(self._table) − 1)       # number 2^x - 1 is often prime

  def __delitem__(self, k):
    j = self._hash_function(k)
    self._bucket_delitem(j, k)                     # may raise KeyError
    self._n −= 1

  def _resize(self, c):                    # resize bucket array to capacity c
    old = list(self.items())               # use iteration to record existing items
    self._table = c * [None]               # then reset table to desired capacity
    self._n = 0                            # n recomputed during subsequent adds
    for (k,v) in old:
      self[k] = v                          # reinsert old key-value pair
```

ABSTRACT HASH TABLE CLASS

```python
class HashMapBase(MapBase):
  """Abstract base class for map using hash-table with MAD compression."""

  def __init__(self, cap=11, p=109345121):
    """Create an empty hash-table map."""
    self._table = cap * [ None ]
    self._n = 0                                  # number of entries in the map
    self._prime = p                              # prime for MAD compression
    self._scale = 1 + randrange(p-1)             # scale from 1 to p-1 for MAD
    self._shift = randrange(p)                   # shift from 0 to p-1 for MAD

  def _hash_function(self, k):
    return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

  def __len__(self):
    return self._n

  def __getitem__(self, k):
    j = self._hash_function(k)
    return self._bucket_getitem(j, k)            # may raise KeyError

  def __setitem__(self, k, v):
    j = self._hash_function(k)
    self._bucket_setitem(j, k, v)                # subroutine maintains self._n
    if self._n > len(self._table) // 2:          # keep load factor <= 0.5
      self._resize(2 * len(self._table) - 1)     # number 2^x - 1 is often prime

  def __delitem__(self, k):
    j = self._hash_function(k)
    self._bucket_delitem(j, k)                   # may raise KeyError
    self._n -= 1

  def _resize(self, c):                   # resize bucket array to capacity c
    old = list(self.items())              # use iteration to record existing items
    self._table = c * [None]              # then reset table to desired capacity
    self._n = 0                           # n recomputed during subsequent adds
    for (k,v) in old:
      self[k] = v                         # reinsert old key-value pair
```

End of Hash Functions

Please proceed to the next video

IT309: Collisions

COLLISION STRATEGIES

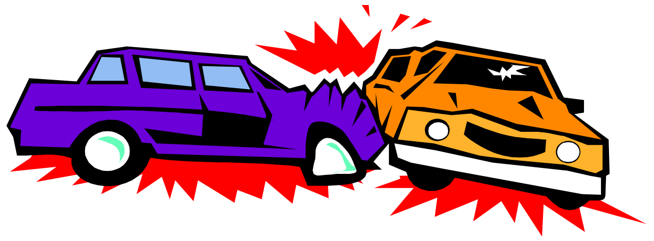
Collisions occur when different elements are mapped to the same cell

Two categories of collision handling strategies:

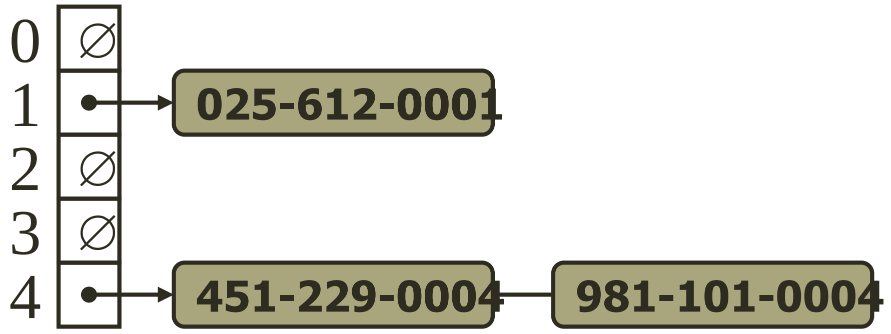
- **Open Addressing** - on collision, probe the table for the next open slot
    - Linear Probing
    - Quadratic Probing
    - Double Hashing (not covered)

- **Separate Chaining** (aka "open hashing")

SEPARATE CHAINING



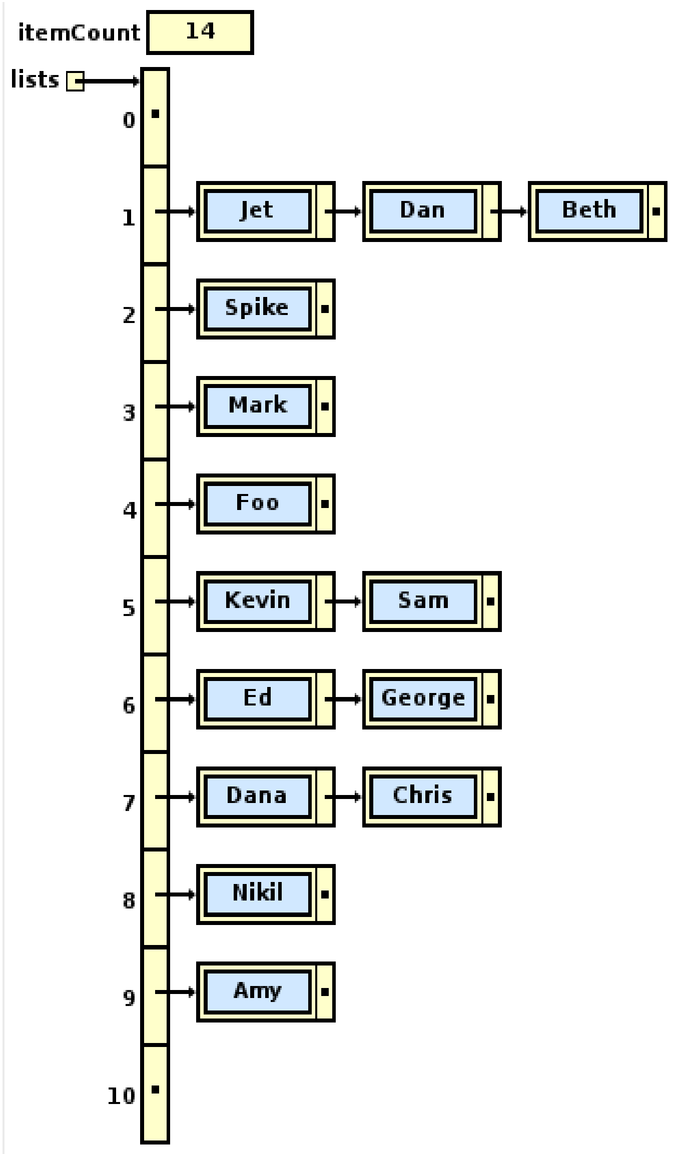
Separate Chaining: let each cell in the table point to a linked list of entries that map there



Separate chaining is simple, but requires additional memory outside the table

SEPARATE CHAINING

- Is an array of linked lists
- Each HT elt. points to the head of a list
- Collisions are added to the list
- “empty slots” point to an empty list
- Load factor can be > 1.0
- Successful search costs are reasonable: 1 + LF/2 probes (LF = table load factor)
- But - successful searches are more expensive than unsuccessful searches for LF < 2.0 (which immediately hashes to an empty slot and thus requires 1 probe)



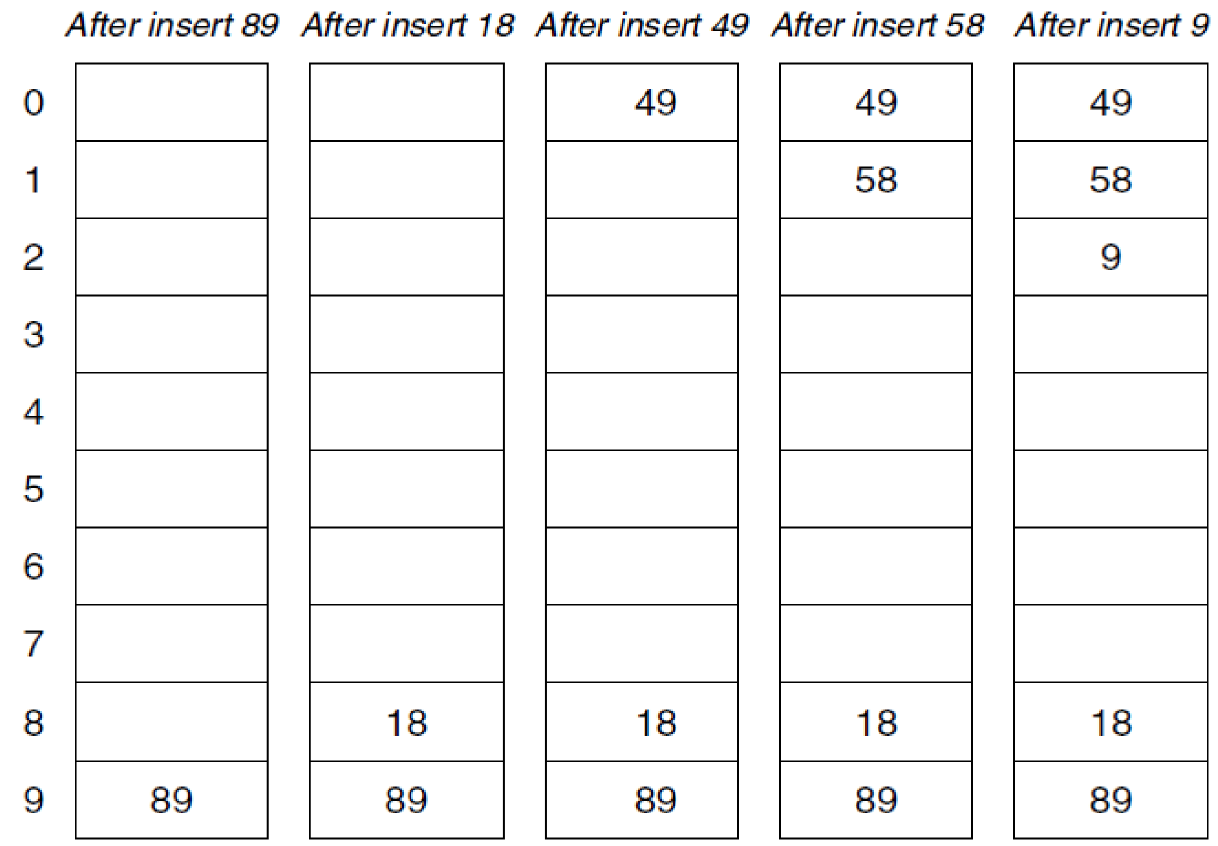
OPEN ADDRESSING

- Do a linear search in the hash table until the next open slot is found, wrap from end to beginning if necessary
  - hashCode(), full?
  - try hashCode()+1, full?
  - try hashCode()+2, full?
  - try hashCode()+3, full? …
- If you return to the original location, the table is full
- Several options on finding the next open slot:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

LINEAR PROBING

(WEISS FIG. 20.5)

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

EXAMPLE

- Insert color names into an array of size 40 (indexed 0 to 39) (wb)
- Hashing function is

  <1st char alphabetic position + 2nd char position> % 40

- Example: brown: b = 2, r = 18, so HF = (2 + 18)%40 = 20
- Range of the output code is 0 to 39
- Linear probing used to handle collisions
- Transactions::
  - Insert in this order: orange, red, green, black, purple, blue, gray, grey
  - Delete: red, blue, green, gray

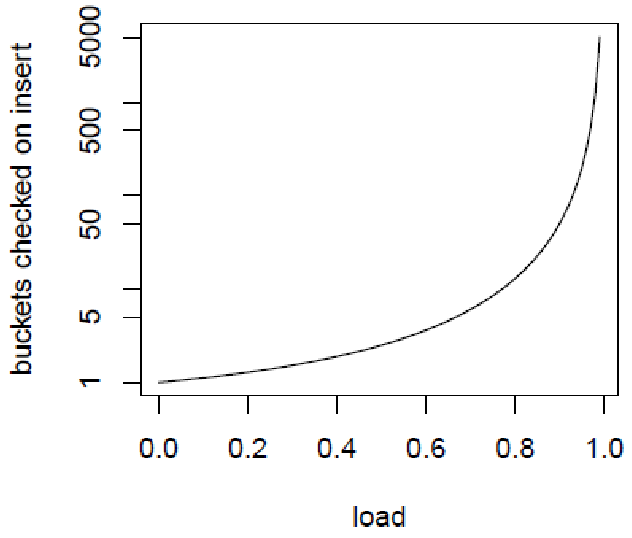
LINEAR PROBING ISSUES

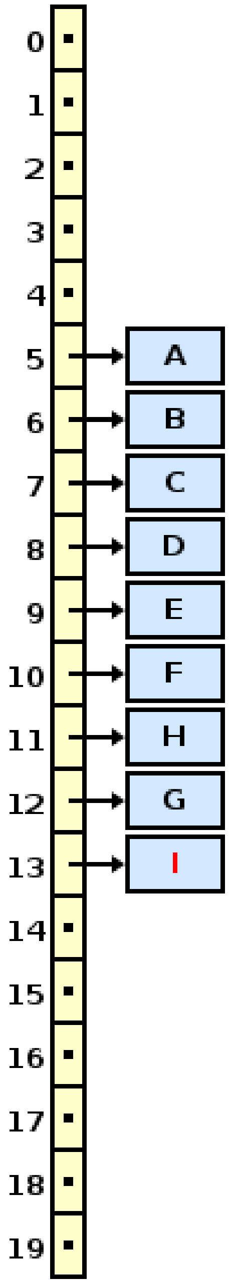
- Most data set codes do not end up being evenly distributed throughout the table
- Adjacent slots are going to fill up and the records tend to occur in clusters or groups
- Clustering is especially bad at high load factors
  - Load factor(λ) ≤ .5 (table half full) reduces clustering

Theorem

The average number of cells examined during insertion with linear probing is

$$\frac{1}{2}\left(1+\frac{1}{(1-load)^2}\right)$$



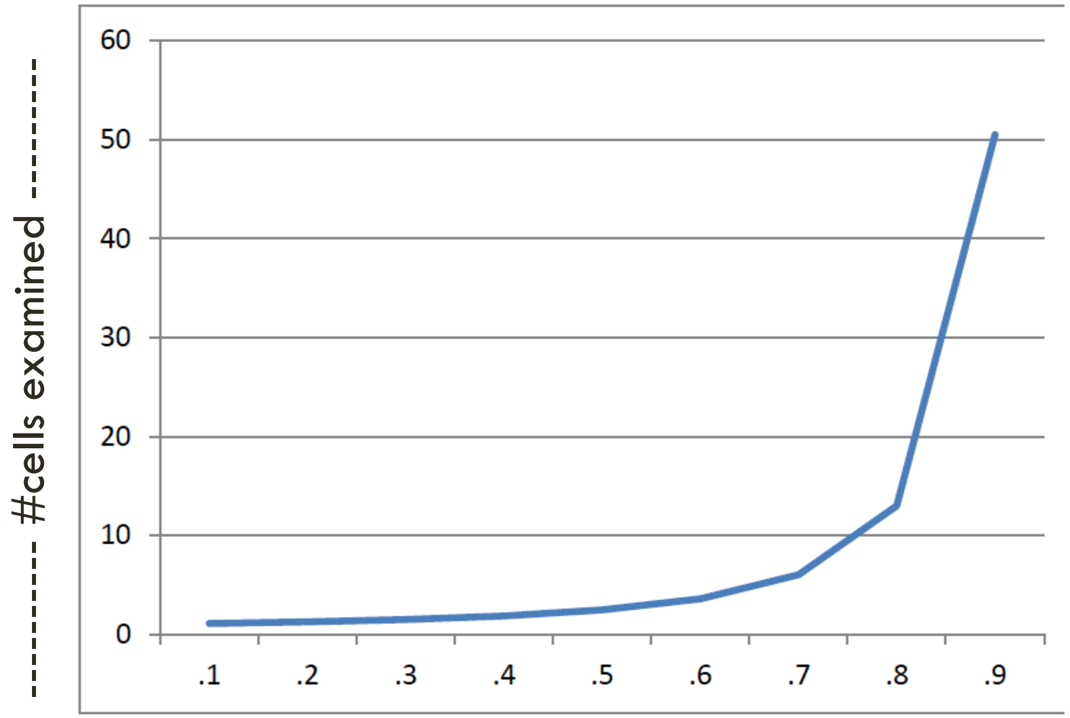


LINEAR PROBING – EFFECT OF LOAD FACTOR

Load factor ≤ .5 (table half full) reduces clustering

λ	#cells examined
.1	1.1
.2	1.3
.3	1.5
.4	1.9
.5	2.5
.6	3.6
.7	6.0
.8	13
.9	50.5

$$\frac{1}{2}\left(1+\frac{1}{(1-load)^2}\right)$$



----------- Load Factor λ -------------

QUADRATIC PROBING

- Insert at:
  - hashCode(), full?
  - try hashCode()+1^2, full?
  - try hashCode()+2^2, full?
  - try hashCode()+3^2, full? …
- Improvement over linear (reduced primary clustering)
- Choose table size that's a prime number and ensure the load factor λ is never more than .5 (50%)
  - then all open slot searches (probes) will be to different locations and an item can always be inserted

QUADRATIC PROBING

(WEISS FIGURE 20.7)

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*	
0			49	49	49	
1						
2				58	58	$\leftarrow 8 + 2^2$
3					9	$\leftarrow 9 + 2^2$
4						
5						
6						
7						
8		18	18	18	18	
9	89	89	89	89	89	

INSERTIONS AND DELETIONS

- With collision probing, deleting an element in the middle of a collision cluster means not finding anything after that element since an unoccupied slot ends the search
- To remedy, add an "occupied" element to each slot to serve as a 'delete' flag for removed elements in the middle of a cluster
- For insert, add the element to the 'delete' slot and remove the flag
- For find or delete, continue searching until the search item or an empty slot is found
- Compress the HT by deleting slots with delete flags during an off-line maintenance window

End of Collisions

Please proceed to the next video

IT309: Hashing Examples

HASH FUNCTION EXAMPLE: SSNS (1/4)

- We want to create a hash table for employee data for a company with 1,000 employees using their U.S. social security numbers (9 digits) as hash function input
- We don't need a data structure with 999,999,999 elements
- We could sum the digits:
  - ABC-DE-FGHI $\rightarrow$ A + B + C + D + E + F + G + H + I
  - Result would be the range 1 to 81, yielding 81 unique values

HASH FUNCTION EXAMPLE: SSNS (2/4)

- Use modulo division with the output ‘tableSize’ as the divisor: ssn%tableSize
- For ssn = 123-45-6789 for various ‘tableSize’, perform modulo arithmetic on the entire ssn value (e.g., 123,456,789%100):

tableSize	Formula	remainder
100	ssn%100	89
200	ssn%200	189
500	ssn%500	289
1000	ssn%1000	789

HASH FUNCTION EXAMPLE: SSNS (3/4)

- Use modulo division with the output ‘tableSize’ as the divisor: ssn%tableSize
- ssn = 123-45-6789 and 321-45-6789 hash to the same values for various ‘tableSize’, resulting in a <u>collision</u>:

tableSize	Formula	Rem: 123-45-6789	Rem: 321-45-6789
100	ssn%100	89	89
200	ssn%200	189	189
500	ssn%500	289	289
1000	ssn%1000	789	789

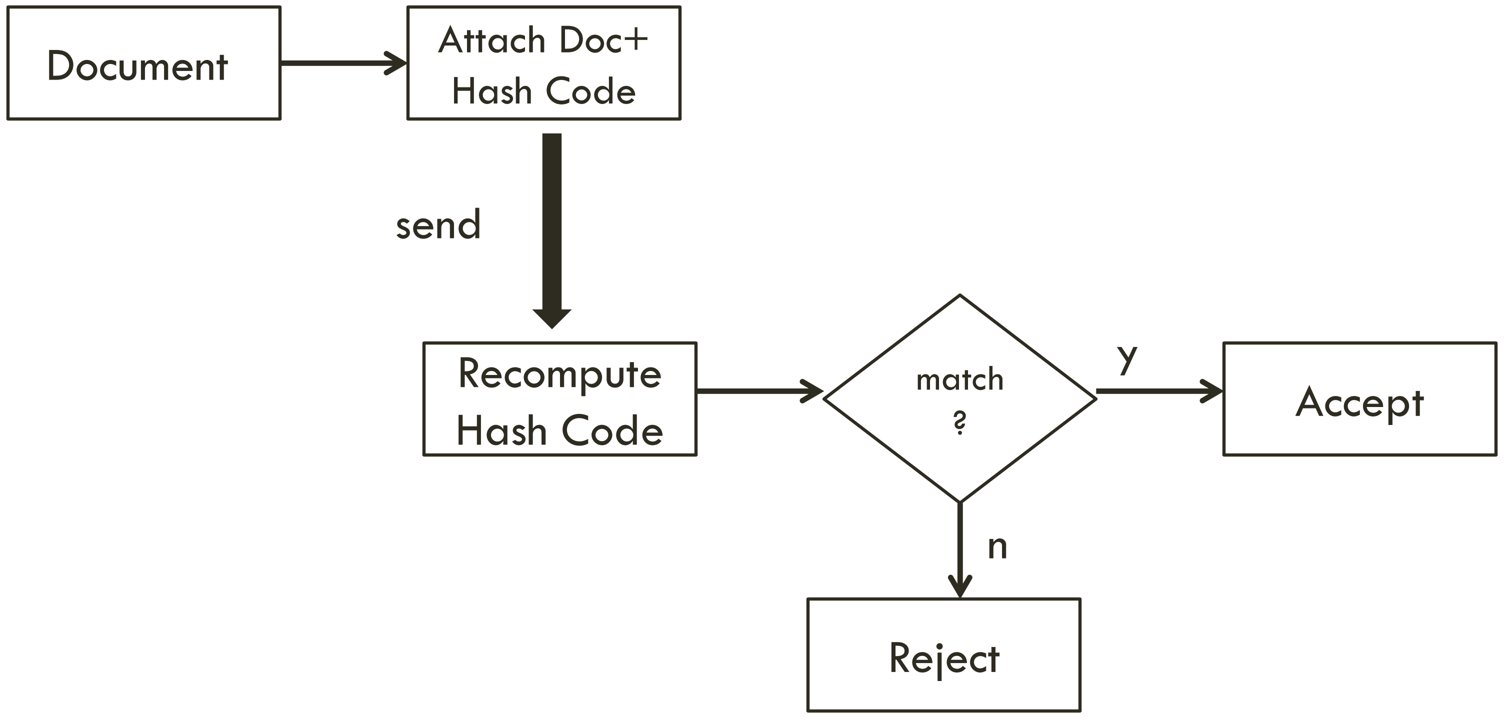
HASH FUNCTION EXAMPLE: SSNS (4/4)

- Use modulo division with the output ‘tableSize’ as the divisor: ssn%tableSize
- Using different divisors can help avoid collisions:

tableSize	Formula	Rem: 123-45-6789	Rem: 321-45-6789
100	ssn%100	89	89
200	ssn%200	189	189
500	ssn%500	289	289
1000	ssn%1000	789	789
2001	ssn%2001	1092	141
2501	ssn%2501	2427	758

HASHING APPLICATIONS

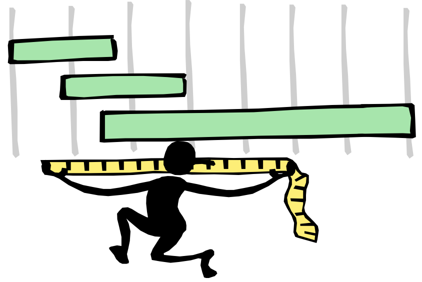
- Data Structure with ~O(1) time complexity
- Security: Digital Signatures, document integrity/authentication (including drivers licenses, passports, etc.)



HASHING APPLICATIONS

- Security: related to Crypto algorithms (e.g., SHA,MD5, DES)
- Algorithm:
  - Hashing: HC = Algorithm(Data)
  - Cryptographic: Code = Algorithm(Data + Key)
- Symmetric vs. asymmetric keys
- This is its own, very sizable, area of study

PERFORMANCE OF HASHING



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%

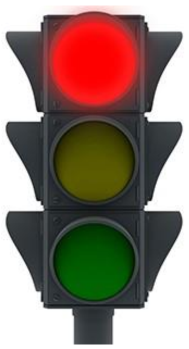
Applications of hash tables:

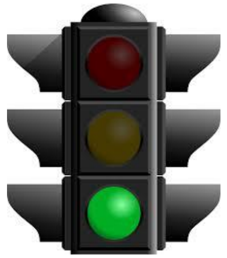
- digital signatures
- encryption
- small databases, compilers
- browser caches

BALANCED TREES VS HASH TABLES

- Balanced Trees
  - Necessary to handle sorted input
  - Harder to implement
  - Supports ordered operations: find minimum, find all keys within some range, etc.
  - Good worst case performance guarantee
- Hash tables
  - Generally easier to implement (separate chaining)
  - Worst case can be poor, but can trade memory to get O(1)
  - If ordered operations not needed, good choice
- Not a huge difference between O(lg N) and O(1)

End of Hashing Examples





Please proceed to the next course activity now or at a later time

Gene Shuman
gshuman@gmu.edu
Department of Computer Science
Volgenau School of Engineering
2219 ENGR
T: 703:993-5214

GEORGE MASON UNIVERSITY

QUESTIONS?