Practical Algorithms

Maps & Hash Tables

Yiannis Giannakopoulos

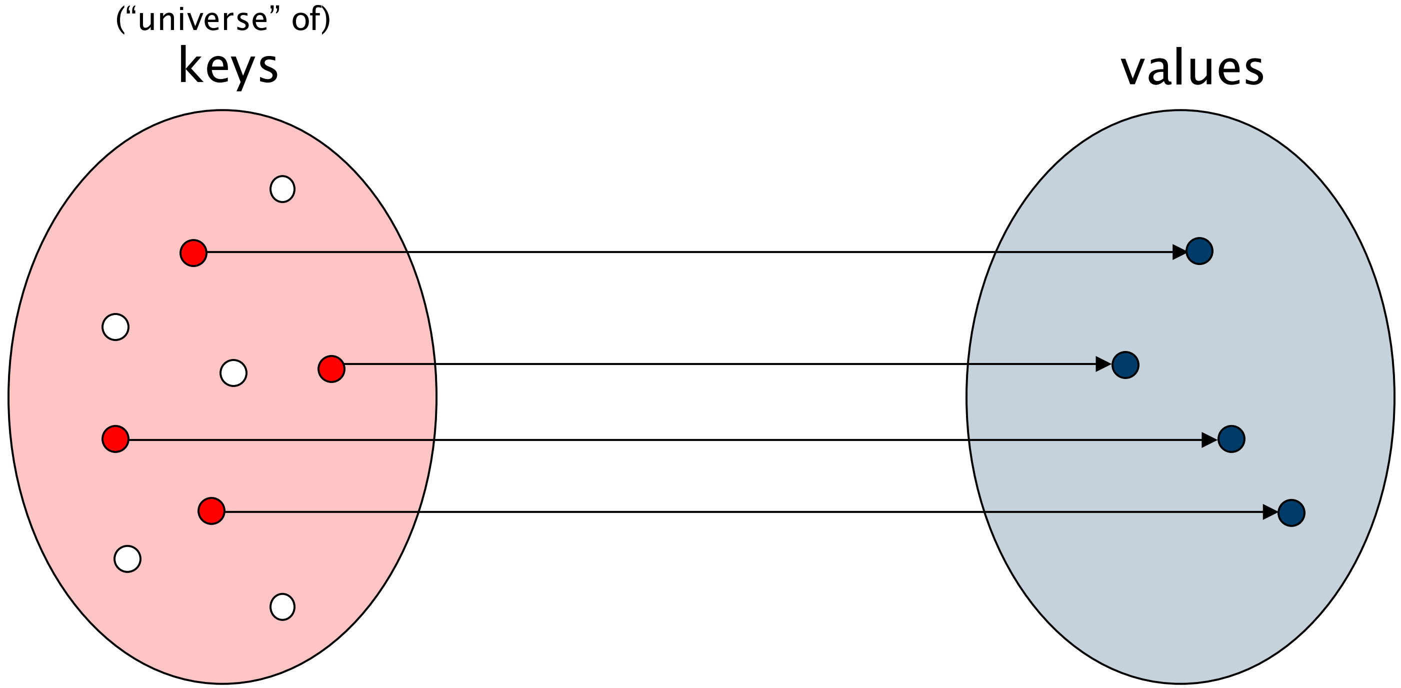
(with thanks to Michele Sevegnani)

School of Computing Science
University of Glasgow

The Map ADT

Dictionaries: Motivation

- Many real-life data sets consist of (key,value) entries
- Examples:
  - (URL, IP address) www.glasgow.ac.uk ☞ 130.209.16.93
  - (student ID, grade)
- Abstraction of a (partial) function:



The Map ADT

- A **map** models a dynamic and searchable collection of **(key,value)** pairs (called **entries** or **elements**)
  - Other names: associative array, dictionary, symbol table, …
  - Multiple entries with the same key are *not* allowed (keys must be unique)

- **Main map operations**
  - INSERT(M,k,v): add an entry (k,v) to map M
  - DELETE(M,k): remove the entry with key k from map M (return NIL if it does not exist)
  - SEARCH(M,k): return the value v of the entry with key k in map M (return NIL if it does not exist)

- **Auxiliary map operation**
  - IS-EMPTY(M): test whether M contains no entries (returns a Boolean value)

Example: ASCII

- **ASCII character encoding (128 entries)**
  - keys: integers {0,1,…,127}, values: characters

0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Example: ASCII

M = {}

- INSERT(M, 65, ‘A’)
- INSERT(M, 71, ‘G’)
- INSERT(M, 113, ‘q’)
- INSERT(M, 109, ‘m’)
- SEARCH(M, 65)
- INSERT(M, 83, ‘S’)
- DELETE(M, 113)
- SEARCH(M, 113)

Example: ASCII

- INSERT(M, 65, ‘A’)
- INSERT(M, 71, ‘G’)
- INSERT(M, 113, ‘q’)
- INSERT(M, 109, ‘m’)
- SEARCH(M, 65)
- INSERT(M, 83, ‘S’)
- DELETE(M, 113)
- SEARCH(M, 113)

M = {}

M = {(65, ‘A’)}

Example: ASCII

- INSERT(M, 65, ‘A’)
- INSERT(M, 71, ‘G’)
- INSERT(M, 113, ‘q’)
- INSERT(M, 109, ‘m’)
- SEARCH(M, 65)
- INSERT(M, 83, ‘S’)
- DELETE(M, 113)
- SEARCH(M, 113)

M = {}

M = {(65, ‘A’)}

M = {(65, ‘A’), (71, ‘G’)}

Example: ASCII

M = {}

- INSERT(M, 65, ‘A’)
- INSERT(M, 71, ‘G’)
- INSERT(M, 113, ‘q’)
- INSERT(M, 109, ‘m’)
- SEARCH(M, 65)
- INSERT(M, 83, ‘S’)
- DELETE(M, 113)
- SEARCH(M, 113)

M = {(65, ‘A’)}

M = {(65, ‘A’), (71, ‘G’)}

M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}

Example: ASCII

- INSERT(M, 65, ‘A’)
- INSERT(M, 71, ‘G’)
- INSERT(M, 113, ‘q’)
- INSERT(M, 109, ‘m’)
- SEARCH(M, 65)
- INSERT(M, 83, ‘S’)
- DELETE(M, 113)
- SEARCH(M, 113)

M = {}

M = {(65, ‘A’)}

M = {(65, ‘A’), (71, ‘G’)}

M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}

M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}

Example: ASCII

$M = \{\}$

- INSERT(M, 65, ‘A’) — $M = \{(65, \text{‘A’})\}$
- INSERT(M, 71, ‘G’) — $M = \{(65, \text{‘A’}), (71, \text{‘G’})\}$
- INSERT(M, 113, ‘q’) — $M = \{(65, \text{‘A’}), (71, \text{‘G’}), (113, \text{‘q’})\}$
- INSERT(M, 109, ‘m’) — $M = \{(65, \text{‘A’}), (71, \text{‘G’}), (113, \text{‘q’}), (109, \text{‘m’})\}$
- SEARCH(M, 65) — return: ‘A’
- INSERT(M, 83, ‘S’)
- DELETE(M, 113)
- SEARCH(M, 113)

Example: ASCII

M = {}

- INSERT(M, 65, ‘A’) — M = {(65, ‘A’)}
- INSERT(M, 71, ‘G’) — M = {(65, ‘A’), (71, ‘G’)}
- INSERT(M, 113, ‘q’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
- INSERT(M, 109, ‘m’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
- SEARCH(M, 65) — return (65, ‘A’)
- INSERT(M, 83, ‘S’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’), (83, ‘S’)}
- DELETE(M, 113)
- SEARCH(M, 113)

Example: ASCII

	M = {}
– INSERT(M, 65, ‘A’)	M = {(65, ‘A’)}
– INSERT(M, 71, ‘G’)	M = {(65, ‘A’), (71, ‘G’)}
– INSERT(M, 113, ‘q’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
– INSERT(M, 109, ‘m’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
– SEARCH(M, 65)	return (65, ‘A’)
– INSERT(M, 83, ‘S’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’), (83, ‘S’)}
– DELETE(M, 113)	M = {(65, ‘A’), (71, ‘G’), (109, ‘m’), (83, ‘S’)}
– SEARCH(M, 113)	

Example: ASCII

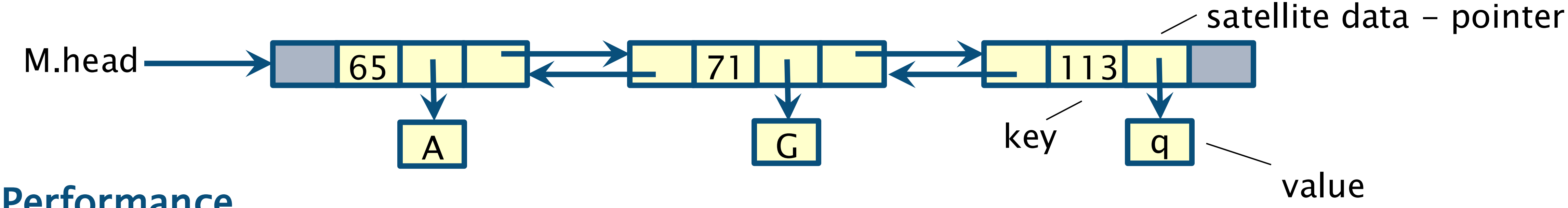
	M = {}
– INSERT(M, 65, ‘A’)	M = {(65, ‘A’)}
– INSERT(M, 71, ‘G’)	M = {(65, ‘A’), (71, ‘G’)}
– INSERT(M, 113, ‘q’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
– INSERT(M, 109, ‘m’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
– SEARCH(M, 65)	return (65, ‘A’)
– INSERT(M, 83, ‘S’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’), (83, ‘S’)}
– DELETE(M, 113)	M = {(65, ‘A’), (71, ‘G’), (109, ‘m’), (83, ‘S’)}
– SEARCH(M, 113)	return NIL

Map Implementations

What is the "best" way to implement such a data structure?

List-based Implementation

- **We can implement a map using a doubly-linked list:**
  - Values are stored as satellite data (attribute if small, pointer for larger structures)



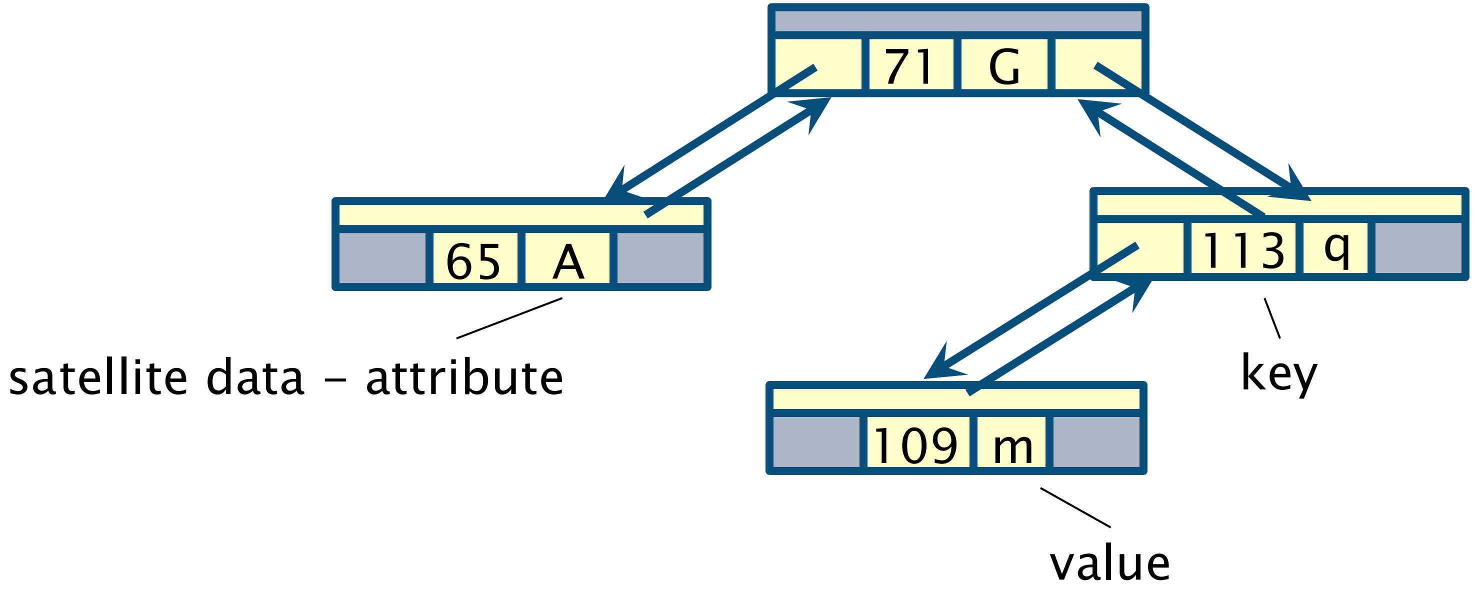
- **Performance**
  - INSERT takes O(1) time (O(n) if we first check for duplicates)
  - SEARCH and DELETE take O(n) – we need to traverse the entire list to look for an entry

- **The list-based implementation is recommended only for maps of *small size***
  - Can we do better?

Tree-based Implementation

- Using a self-balancing trees we can guarantee a worst-case running time of:
  - O(log n) for all the main map ADT operations

- Additionally, an in-order traversal allows us to get a sorted sequence of all the pairs stored in the map

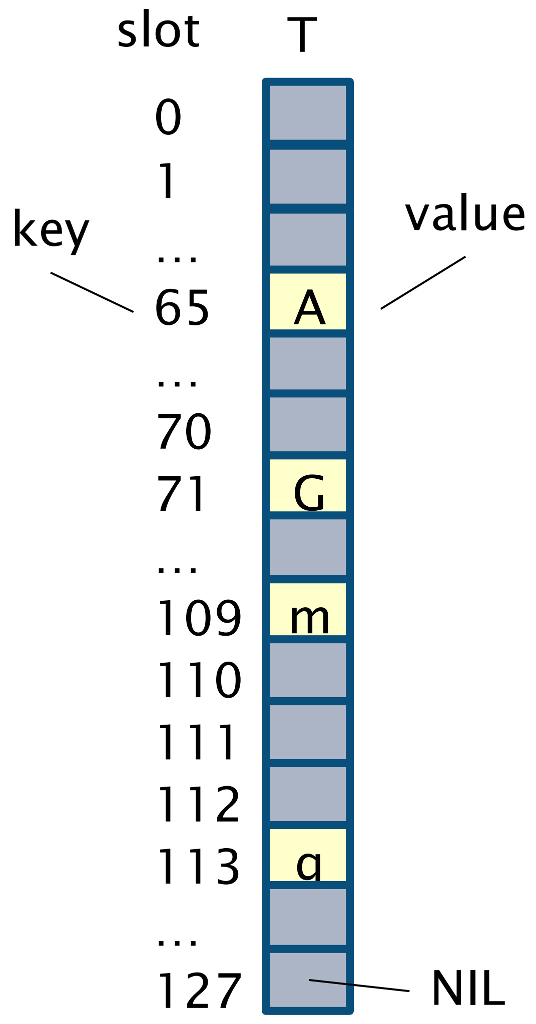


- Can we do even better?

Direct-address Tables

- **Assumptions**
  - Each element of our map M has an integer key drawn from the universe U = {0,1,…, m – 1}
  - Recall: no two elements have the same key

- **A direct-address table is an array T[0,..,m – 1] that can represent map M in the following way:**
  - Each position (also called slot or bucket) in T corresponds to a key in the universe U
  - Slot k contains/points to the value of the element M with key k
    - In other words: if (k,v)∈M then T[k]=v.
  - If no element has key k, then T[k] = NIL



Direct-address Table: Map Implementation

- **Operations are trivial to implement**
  - Each operation takes O(1) time

element x = (key,value)

```
DIRECT-ADDRESS-INSERT(T,x)
  T[x.key] = x.value
```

- **However, what if:**
  1. The keys are not natural numbers?
  2. The universe U is much larger than the *"actual"* number of keys that we are expecting to use, i.e $|U| \gg m$?

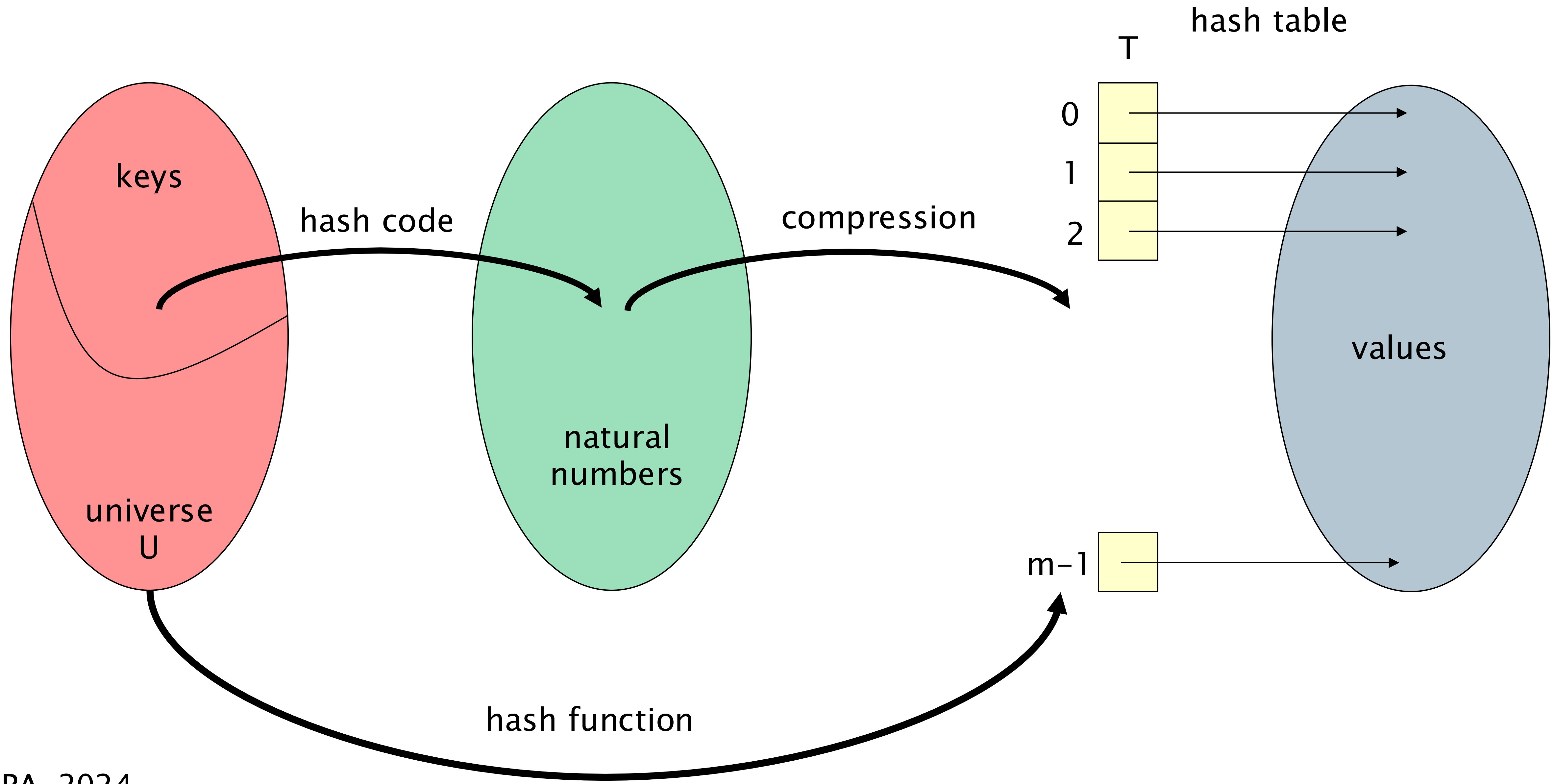
```
DIRECT-ADDRESS-SEARCH(T,k)
  return T[k]
```

- **Hashing deals with issues 1. and 2. by:**
  1. Encoding
  2. Compression

```
DIRECT-ADDRESS-DELETE(T,k)
  T[k] = NIL
```

Hash Tables

Hashing: Overview

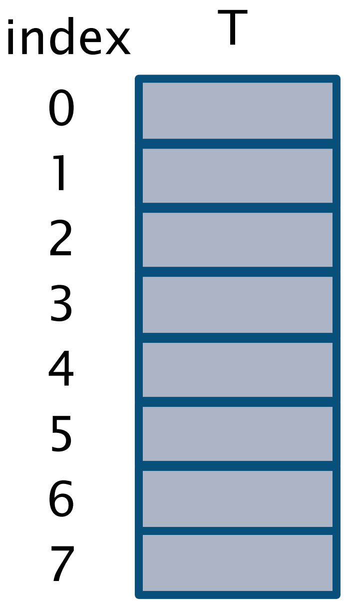


The Hash Table Data Structure

- Generalizes direct-address tables by adding a hash function
- Consists of:
  1. An array $T[0,..,m-1]$ of fixed size m (called hash table or bucket array)
  2. A hash function $h: U \rightarrow \{0,1,\ldots,m-1\}$ mapping keys to slots of T
- Hash collision: when two keys are mapped to the same slot of the hash table
  - That is, $h(k_1)=h(k_2)$ for $k_1 \neq k_2$ (where $k_1, k_2 \in U$)
  - In general, hash collisions are <u>unavoidable</u> (if $|U| > m$)
  - <u>However:</u> a "good" hash function spreads keys as "evenly" as possible over the slots of T
    - ➢ Each backet should be used with equal probability for data randomly sampled from the universe U
- Additionally: hash functions should be "simple" and fast to compute
- Under the above assumptions: hash tables support INSERT, DELETE and SEARCH operations in $O(1)$ time "on average"

Example: ASCII

- ASCII table with hash function h(k) = k mod 8
  - U = {0,1,…,127} and size of hash table T is m = 8
- Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’) in hash table T



Example: ASCII

- ASCII table with hash function **h(k) = k mod 8**
  - U = {0,1,…,127} and size of hash table T is m = 8
- **Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)**

index	T	
0		
1	65	A
2		
3		
4		
5		
6		
7		

- INSERT(T, (65, ‘A’))
- h(65) = 65 mod 8 = 1
- Insert element in slot 1

Example

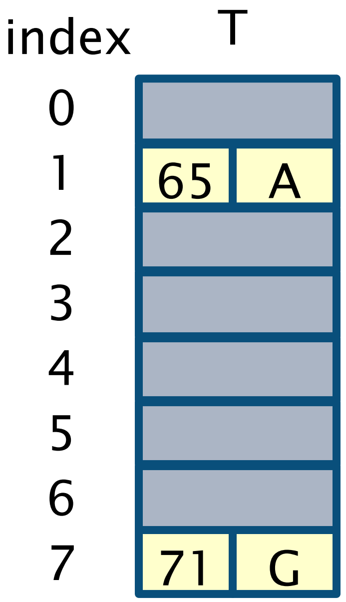
- ASCII table with hash function h(k) = k mod 8
  - U = {0,…127} and size of hash table T is m = 8
- Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)

index	T	
0		
1	65	A
2		
3		
4		
5		
6		
7	71	G

- INSERT(T, (71, ‘G’))
- h(71) = 71 mod 8 = 7
- Insert element in slot 7

Example

- ASCII table with hash function h(k) = k mod 8
  - U = {0,...127} and size of hash table T is m = 8
- Insert (65, 'A'), (71, 'G'), (113, 'q'), (109, 'm') and (83, 'S')



- INSERT(T, (113, 'q'))
- h(113) = 113 mod 8 = 1

- Insert element in slot 1, but slot 1 is already occupied
- We say that keys 65 and 113 collide

Example

- ASCII table with hash function h(k) = k mod 8
  - U = {0,…127} and size of hash table T is m = 8
- Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)

index	T	
0		
1	113	q
2		
3		
4		
5		
6		
7	71	G

- INSERT(T, (113, ‘q’))
- h(113) = 113 mod 8 = 1
- A (bad, in general) strategy to resolve collisions is to store only the most recent key/value
- We will study more sophisticated strategies to resolve collisions later in these lectures

Example

- ASCII table with hash function $h(k) = k \bmod 8$
  - $U = \{0, \ldots 127\}$ and size of hash table T is $m = 8$
- Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)

index	T	
0		
1	113	q
2		
3		
4		
5	109	m
6		
7	71	G

- INSERT(T, (109, ‘m’))
- $h(109) = 109 \bmod 8 = 5$
- Insert element in slot 5

Example

- **ASCII table with hash function h(k) = k mod 8**
  - U = {0,...127} and size of hash table T is m = 8
- **Insert (65, 'A'), (71, 'G'), (113, 'q'), (109, 'm') and (83, 'S')**

index	T	
0		
1	113	q
2		
3	83	S
4		
5	109	m
6		
7	71	G

- INSERT(T, (83, 'S'))
- h(83) = 83 mod 8 = 3
- Insert element in slot 3

Encoding

Encoding general keys as natural numbers

- Most hash functions operate on natural numbers, ie they assume as a universe of keys $U = \mathbb{N}$

- There are several methods (called hash codes) to convert/encode an arbitrary object as a natural number, e.g.
  - Integer casting
  - Component sum
  - Memory address
  - Polynomial hashing

- Here we will only describe integer casting and component sum here, *very briefly*.

Integer Casting

- Most data types have a "natural" bit representation, in every programming language
  - So, we can use as key the integer corresponding to that binary number
  - Example: $10011_2 = 19_{10}$
- For example: Python uses 64-bit values to encode many fundamental types, e.g. `float` and `int`
  - So, integer casting can be readily used for types:
- For longer types, e.g. strings, we need to perform some kind of "merging"
  - For example, an object $(x_0, x_1, \ldots, x_{n-1})$ where all x_i are 64-bit integers can be represented as
    - $\sum_{i=0}^{n-1} x_i$, or
    - $x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$, where $\oplus$ is the XOR operator
  - ➢ This is known as component sum hashing

Compression: Hash Functions

$$\mathbb{N} \longrightarrow \{0,1,\ldots,m-1\}$$

Truncation

- **Take the first/last few digits of the key**
  - Problem: it may generate many collisions if there are regularities in the input keys

- **Example**
  - Student IDs consisting of 8 digits: 2023 1734
  - Numbers are assigned sequentially
  - Students in a given class/year will tend to have IDs close together, and all beginning with the same first few digits
  - But: taking the <u>last</u> three digits will work a lot better!

Division

- Map a key k into one of m slots by taking the remainder of the division of k by m
  - The hash function is $h(k) = k \bmod m$
    - Python: `k % m`
- Good practice: to ensure that data is distributed fairly, we usually choose the table size m to be
  - Prime
  - Not “too close” to an exact power of 2
- If $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k
  - Examples: $101011_2 \% 10_2 = \mathbf{1}_2$, $101011_2 \% 100_2 = \mathbf{11}_2$
  - The analogous case for decimal numbers would be division by powers of 10:

    $234_{10} \% 10_{10} = \mathbf{4}_{10}$, $234_{10} \% 100_{10} = \mathbf{34}_{10}$
- If use of lower-order bits is suitable, better to simply truncate

Example: Hashing by Division

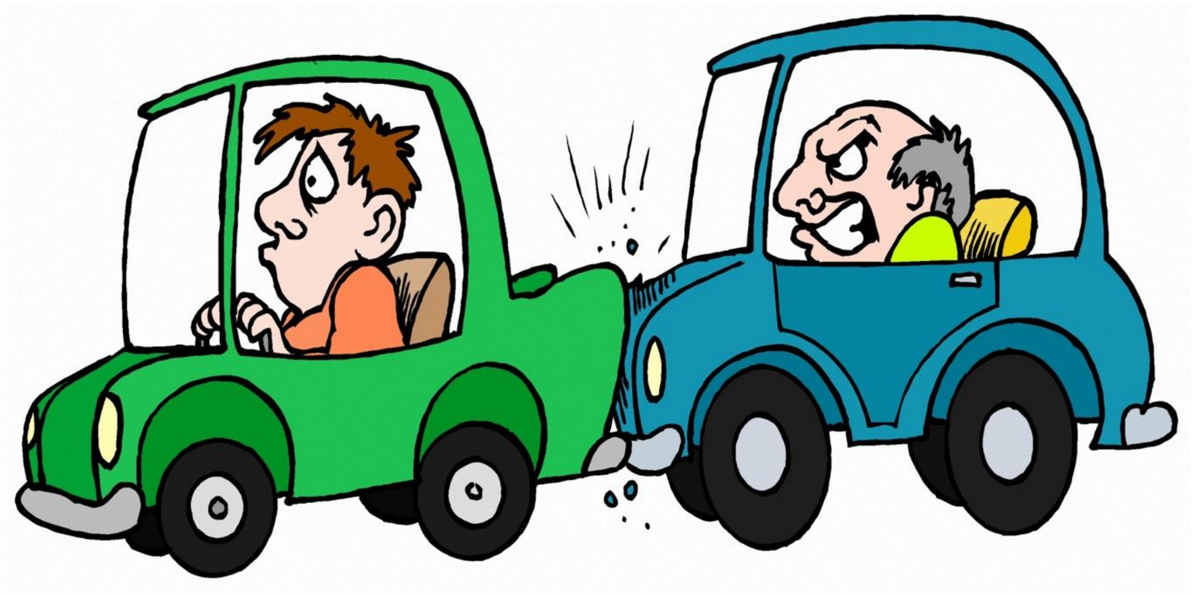
- Suppose we want to allocate a hash table to hold roughly 5000 keys

- We pick m to be a prime close to 5000 but *not* near *any* power of 2
  - $2^{12} = 4069$
  - $2^{13} = 8192$

- Primes near 5000:
  - 4987, 4993, 4999, 5003, 5009, 5011

- So, our hash function could be $h(k) = k \bmod 5003$

Collision Resolution



Chaining

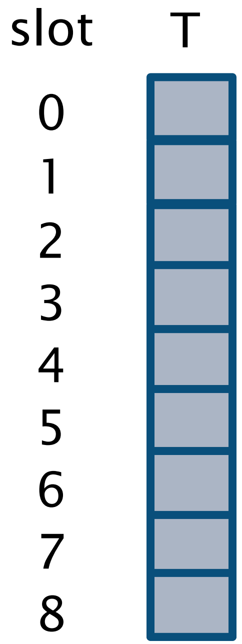
Collision Resolution by Chaining

- Each slot of the hash table points to its own (doubly) linked list (called chain)
- All elements that hash to the same slot are stored in that slot's list
  - List T[i] holds elements (k,v) for which h(k)=i, i=0,1,…,m−1
- Example: ASCII with m=9
  - U = {0, …,127}
  - Insert key sequence: 122, 71, 75, 37, 65, 109
  - Assume that we use a hash function h: U → {0,…,8} such that:

$$h(37) = h(65) = h(122) = 3$$

$$h(71) = 6$$

$$h(75) = h(109) = 8$$



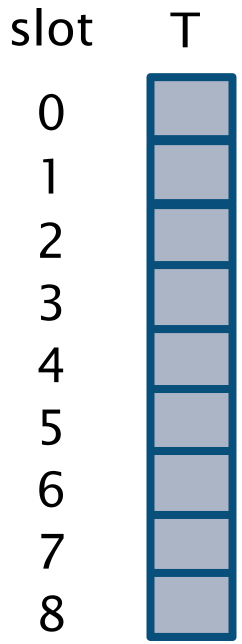
Collision Resolution by Chaining

- Each slot of the hash table points to its own (doubly) linked list (called chain)
- All elements that hash to the same slot are stored in that slot's list
  - List T[i] holds elements (k,v) for which h(k)=i,   i=0,1,…,m−1
- Example: ASCII with m=9
  - U = {0, …,127}
  - Insert key sequence: 122, 71, 75, 37, 65, 109
  - Assume that we use a hash function h: U → {0,…,8} such that:

$$h(37) = h(65) = h(122) = 3$$

$$h(71) = 6$$

$$h(75) = h(109) = 8$$



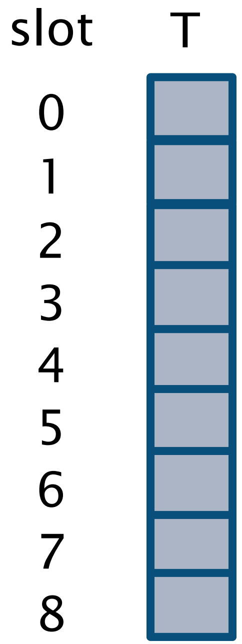
Collision Resolution by Chaining

- Each slot of the hash table points to its own (doubly) linked list (called chain)
- All elements that hash to the same slot are stored in that slot's list
  - List T[i] holds elements (k,v) for which h(k)=i,   i=0,1,…,m−1
- Example: ASCII with m=9
  - U = {0, …,127}
  - Insert key sequence: 122, 71, 75, 37, 65, 109
  - Assume that we use a hash function h: U → {0,…,8} such that:

$$h(37) = h(65) = h(122) = 3$$

$$h(71) = 6$$

$$h(75) = h(109) = 8$$



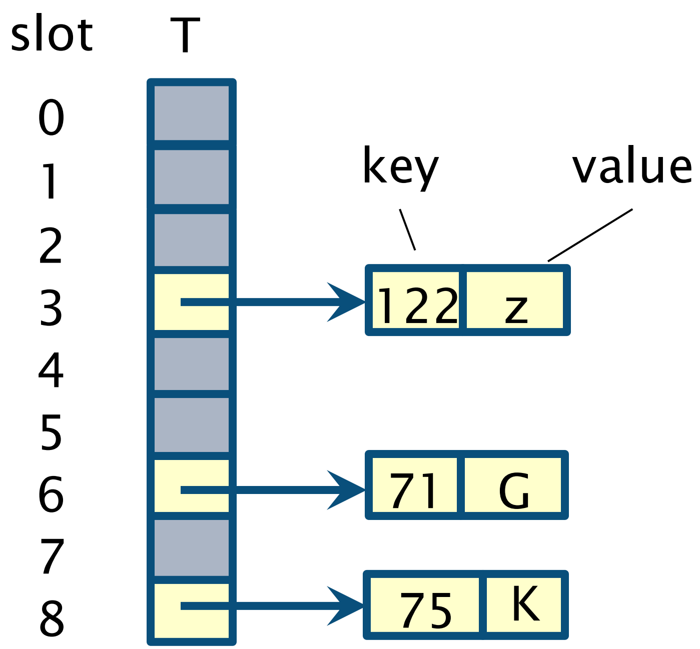
Collision Resolution by Chaining

- Each slot of the hash table points to its own (doubly) linked list (called chain)
- All elements that hash to the same slot are stored in that slot's list
  - List $T[i]$ holds elements (k,v) for which $h(k)=i$, $i=0,1,\ldots,m-1$
- Example: ASCII with m=9
  - $U = \{0, \ldots, 127\}$
  - Insert key sequence: 122, 71, 75, 37, 65, 109
  - Assume that we use a hash function $h: U \rightarrow \{0,\ldots,8\}$ such that:

$$h(37) = h(65) = h(122) = 3$$

$$h(71) = 6$$

$$h(75) = h(109) = 8$$



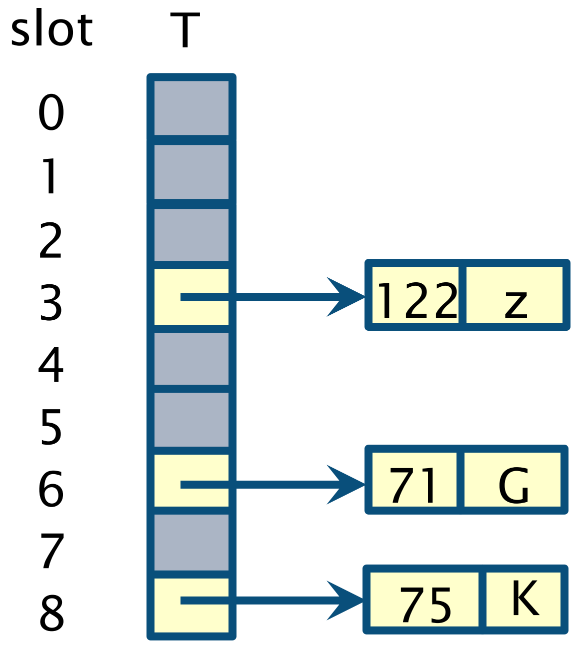
Collision Resolution by Chaining

- Each slot of the hash table points to its own (doubly) linked list (called chain)
- All elements that hash to the same slot are stored in that slot's list
  - List T[i] holds elements (k,v) for which h(k)=i,   i=0,1,…,m−1
- Example: ASCII with m=9
  - U = {0, …,127}
  - Insert key sequence: 122, 71, 75, 37, 65, 109
  - Assume that we use a hash function h: U → {0,…,8} such that:

$$h(37) = h(65) = h(122) = 3$$

$$h(71) = 6$$

$$h(75) = h(109) = 8$$



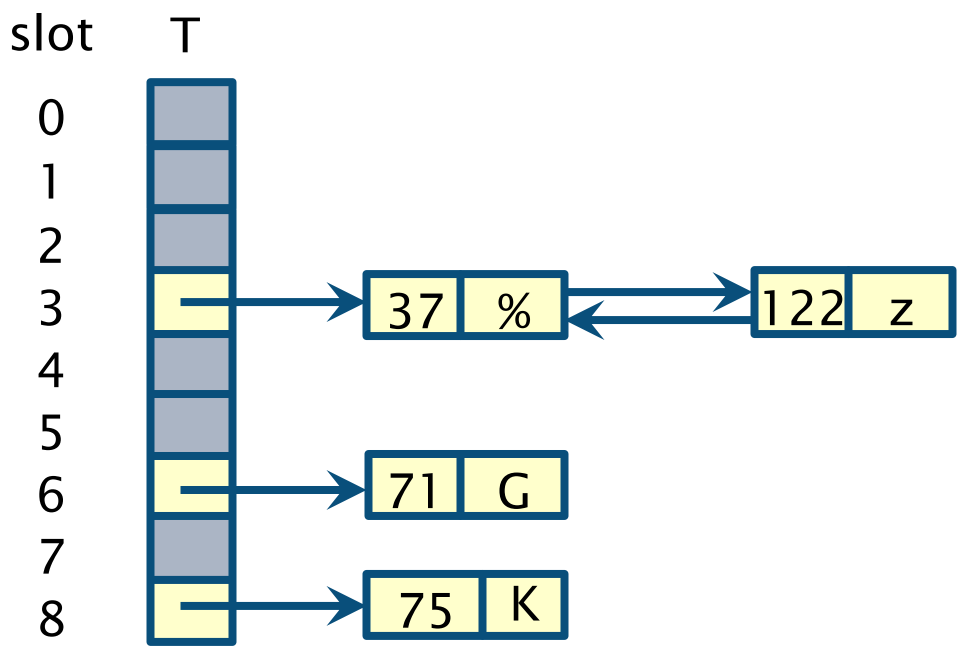
Collision Resolution by Chaining

- Each slot of the hash table points to its own (doubly) linked list (called chain)
- All elements that hash to the same slot are stored in that slot's list
  - List T[i] holds elements (k,v) for which h(k)=i, i=0,1,…,m−1
- Example: ASCII with m=9
  - U = {0, …,127}
  - Insert key sequence: 122, 71, 75, 37, 65, 109
  - Assume that we use a hash function h: U → {0,…,8} such that:

$$h(37) = h(65) = h(122) = 3$$

$$h(71) = 6$$

$$h(75) = h(109) = 8$$



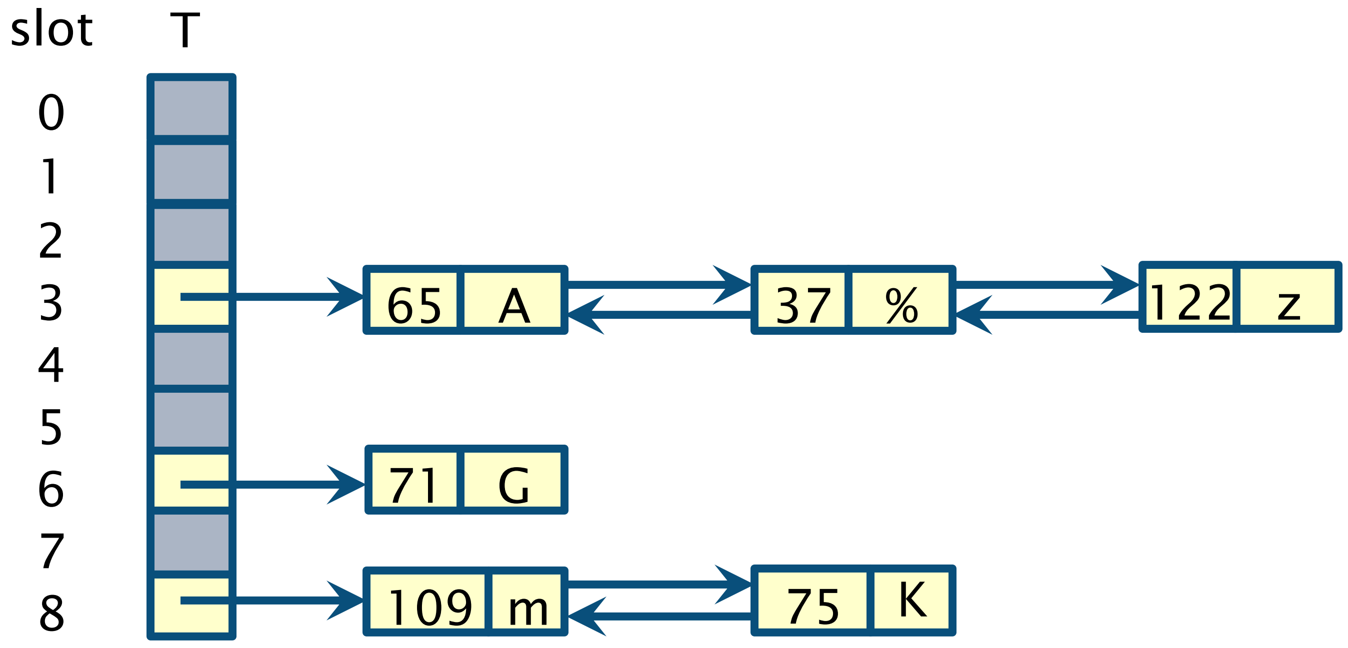
Collision Resolution by Chaining

- Each slot of the hash table points to its own (doubly) linked list (called chain)
- All elements that hash to the same slot are stored in that slot's list
  - List T[i] holds elements (k,v) for which h(k)=i, i=0,1,…,m−1
- Example: ASCII with m=9
  - U = {0, …,127}
  - Insert key sequence: 122, 71, 75, 37, 65, 109
  - Assume that we use a hash function h: U → {0,…,8} such that:

$$h(37) = h(65) = h(122) = 3$$

$$h(71) = 6$$

$$h(75) = h(109) = 8$$



Open-Address Hashing

Collision Resolution by Open Addressing

- **General scheme: if a collision occurs, an alternative cell is tried (or "probed") until an empty cell is found**
  - Appropriate when memory availability is limited, and we cannot use auxiliary data structures (like linked lists in chaining)
  - The load factor needs to be at most $\alpha \leq 1$: otherwise, we may overflow the hash table

- **Rigorously, open addressing can be modelled by adding an extra parameter to our hash function:**

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

where $h(k,i)$ gives the slot that we should probe at our i-th try.
  - Implicit assumption: each key should probe <u>all</u> slots
  - Formally, this means that ($h(k,0)$, $h(k,1)$, …, $h(k,m-1)$) is a permutation of $(0, 1,\ldots, m-1)$, for all $k \in U$

Open Addressing: Insertion

- For a given hash function h(k,i), the HASH-INSERT procedure takes as input a hash table T and a key k and
  - Returns the slot number where it stores k, or
  - Raises an error because T is already full

```
HASH-INSERT(T,k)
  i = 0
  while i < m
    j = h(k,i)
    if T[j] == NIL
       T[j] = k
       return j
    else i = i + 1
  error “hash table overflow”
```

Example: Linear Probing

- Hashing by division into a table of size $m=8$
- Open addressing by sequentially probing slot $i+1$ after slot i (wrapping around when $i = m$)
  - i.e. $h(k,i) = (k+i) \bmod 8$

```
HASH-INSERT(T,k)
  i = 0
  while i < m
    j = h(k,i)
    if T[j] == NIL
       T[j] = k
       return j
    else i = i + 1
  error “hash table overflow”
```

slot	T
0	
1	113
2	
3	83
4	
5	109
6	
7	71

Example: Linear Probing

- Hashing by division into a table of size $m=8$
- Open addressing by sequentially probing slot $i+1$ after slot i (wrapping around when $i = m$)
  - i.e. $h(k,i) = (k+i) \bmod 8$

```
HASH-INSERT(T,k)
  i = 0
  while i < m
    j = (k+i) % m
    if T[j] == NIL
       T[j] = k
       return j
    else i = i + 1
  error "hash table overflow"
```

slot	T
0	
1	113
2	
3	83
4	
5	109
6	
7	71

- INSERT(T, 65)
- $h(65) = 65 \bmod 8 = 1$
- Collision

Example: Insertion

- Hashing by division into a table of size m=8
- Open addressing by sequentially probing slot i+1 after slot i (wrapping around when i = m)
  - i.e. h(k,i) = (k+i) mod 8

```
HASH-INSERT(T,k)
  i = 0
  while i < m
    j = h(k,i)
    if T[j] == NIL
       T[j] = k
       return j
    else i = i + 1
  error “hash table overflow”
```

slot	T
0	
1	113
2	65
3	83
4	
5	109
6	
7	71

- INSERT(T, 65)
- h(65) = 65 mod 8 = 1
- Insert element in slot 2

Example: Insertion

- Hashing by division into a table of size m=8
- Open addressing by sequentially probing slot i+1 after slot i (wrapping around when i = m)
  - i.e. h(k,i) = (k+i) mod 8

```
HASH-INSERT(T,k)
  i = 0
  while i < m
    j = h(k,i)
    if T[j] == NIL
       T[j] = k
       return j
    else i = i + 1
  error “hash table overflow”
```

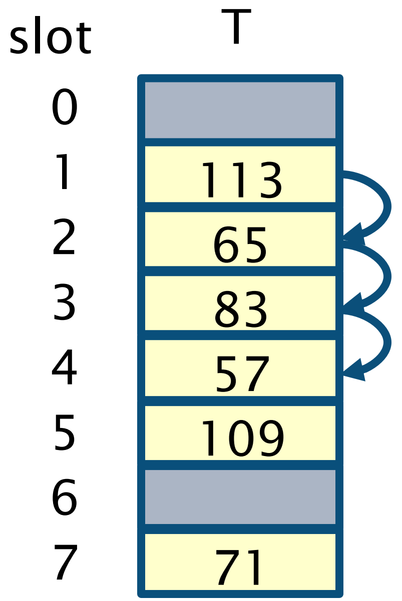
slot	T
0	
1	113
2	65
3	83
4	
5	109
6	
7	71

- INSERT(T, 57)
- h(57) = 57 mod 8 = 1
- Collision

Example: Insertion

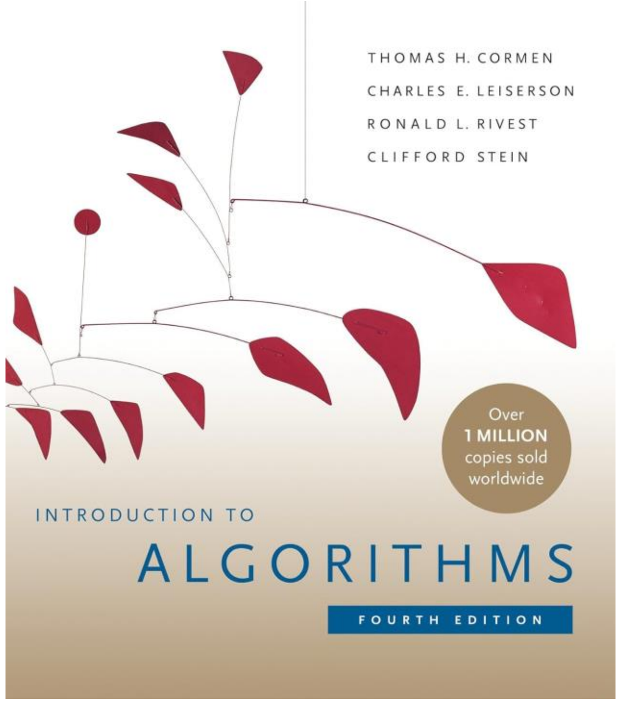
- Hashing by division into a table of size m=8
- Open addressing by sequentially probing slot i+1 after slot i (wrapping around when i = m)
  - i.e. h(k,i) = (k+i) mod 8

```
HASH-INSERT(T,k)
  i = 0
  while i < m
    j = h(k,i)
    if T[j] == NIL
       T[j] = k
       return j
    else i = i + 1
  error "hash table overflow"
```



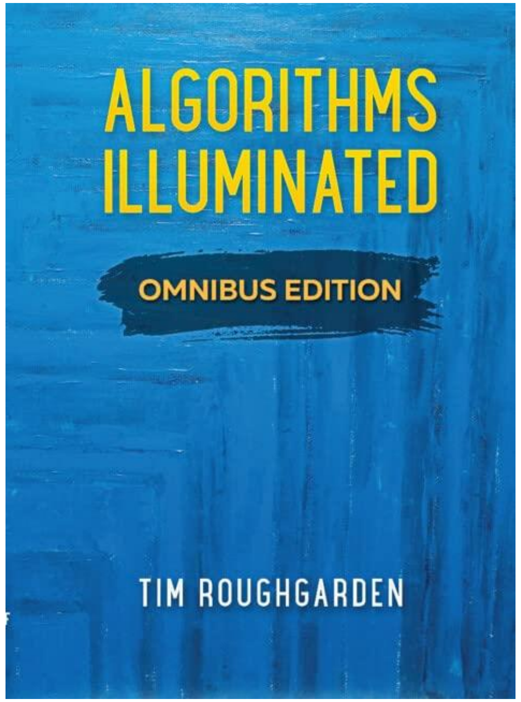
- INSERT(T, 57)
- h(57) = 57 mod 8 = 1
- Insert element in slot 4

Hashing: Further Reading



"Introduction to Algorithms"
(4th edition)
by **Cormen, Leiserson, Rivest, and Stein**

Chapter 11



"Algorithms Illuminated"
(Omnibus edition)
by **Roughgarden**

Chapter 12