[CSED233-01] Data Structure

Dictionary & Hashing

Jaesik Park

POSTECH

f

Dictionary

- A set Abstract Data Types (ADT) of pairs
  - x = (*key*, *info*)
  - Each possible key appears just once in the set – unique ID

- Fundamental operations:
  - *Insert(x, D)* – to store/put x into D
  - *Delete(k, D)* – to remove x = (k, *info*) from D
  - *Search(k, D)* – to lookup x = (k, *info*) from D
    - Returns the *info* (if any) that is bound to a given key k
  - *Etc.*

Dictionary Implementations

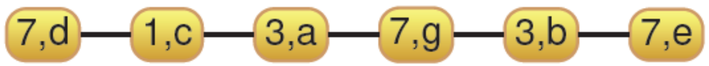
- Three approaches to *Search Problem*
  - Sequential methods – Sorted list
  - *Hashing* method – direct access by *key* values
  - *Tree indexing* methods

Data Structure	Worst	Average
Unsorted list	$O(n)$	
Sorted list	$O(\log n)$	
Hash Table	$O(n)$	$O(1)$
Binary Search Tree	$O(n)$	$O(\log n)$
Balanced Search Trees (AVL, 2-3)	$O(\log n)$	

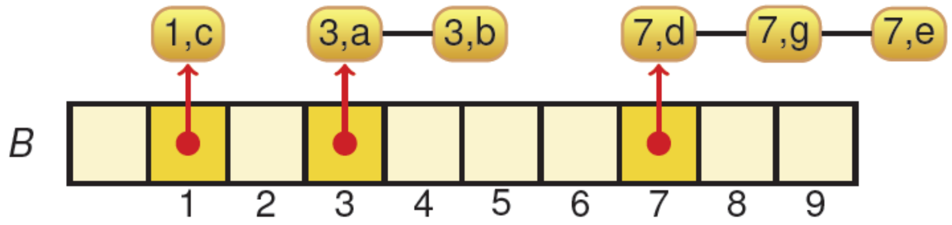
Review: Bucket Sort

- Non-comparison sort
  - Phase 1: scattering keys into a number of buckets
    - If you need to sort a single bucket list, sort each non-empty bucket (either recursively or using a different sorting algorithm, e.g., insertion sort)
  - Phase 2: gathering
    - Visit the buckets in order & empty them into the original list
- Simple example:
  - A list of n (key, info) pairs with key range $[0, N\text{-}1]$

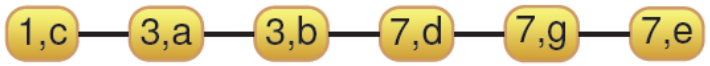
Review: Bucket Sort



- Phase 1: scattering into buckets → O(n)



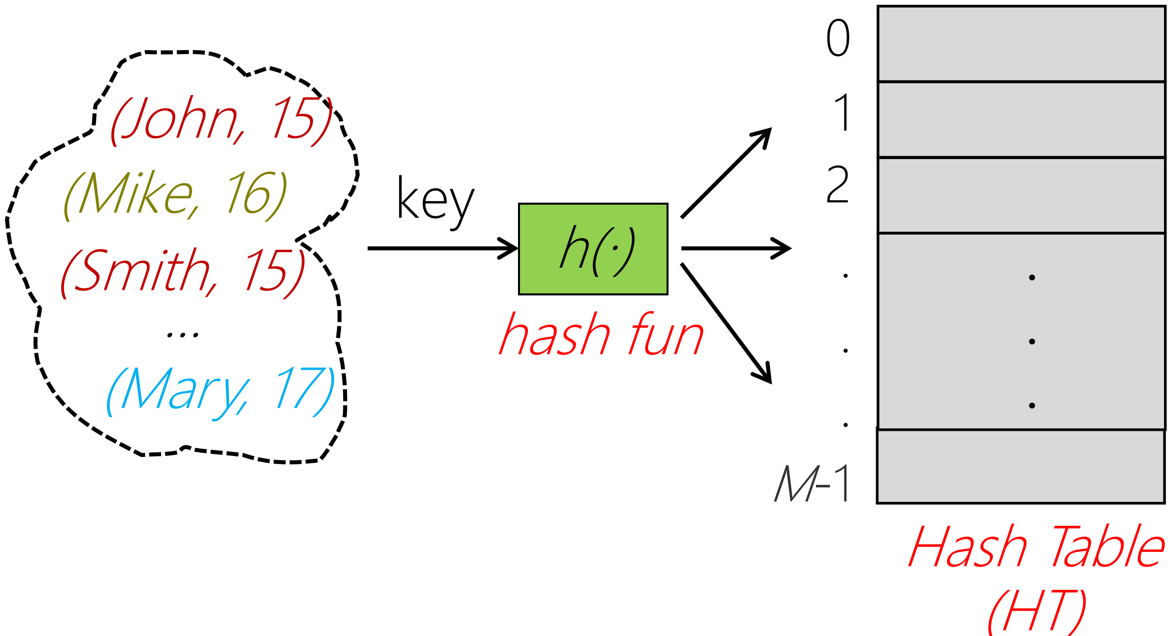
- Phase 2: Gathering → O(n + N)



- O(n + N) time in the average case
- Efficient
  - if keys come from a small interval [0, N - 1]

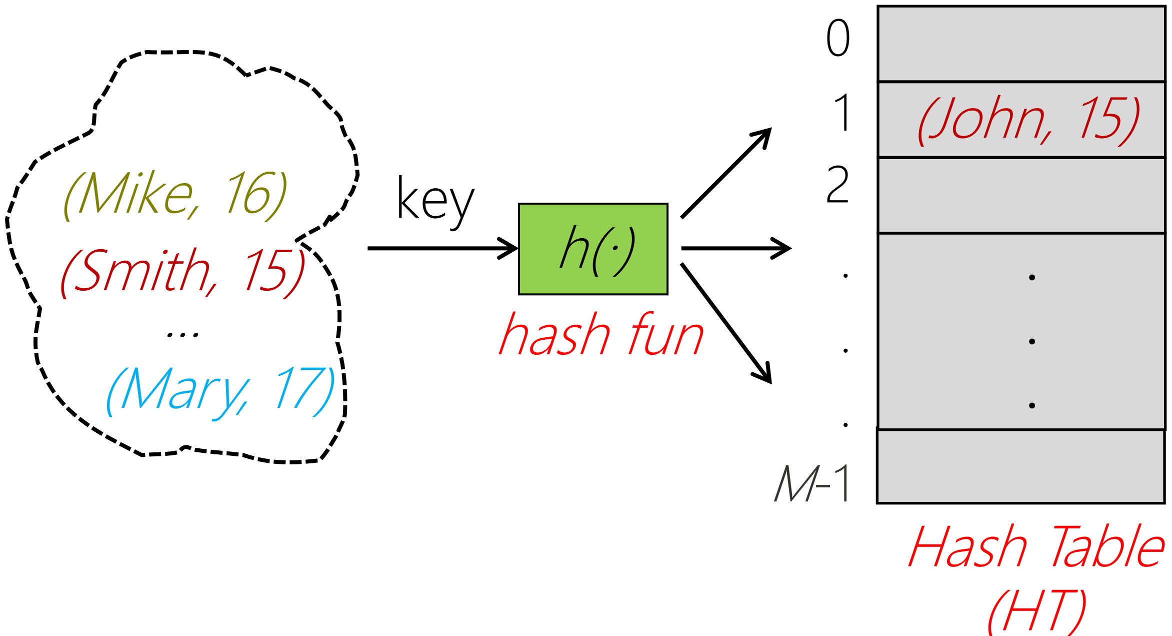
Hashing

- Mapping a key value to a position in a hash table (HT)
  - Hash table (array) *HT[0..M-1]*
    - Each position in *HT* is known as a slot
    - A slot can normally hold only one pair (*key*, *info*)
  - Hash function *h(·): a set of keys → [0..M-1]*



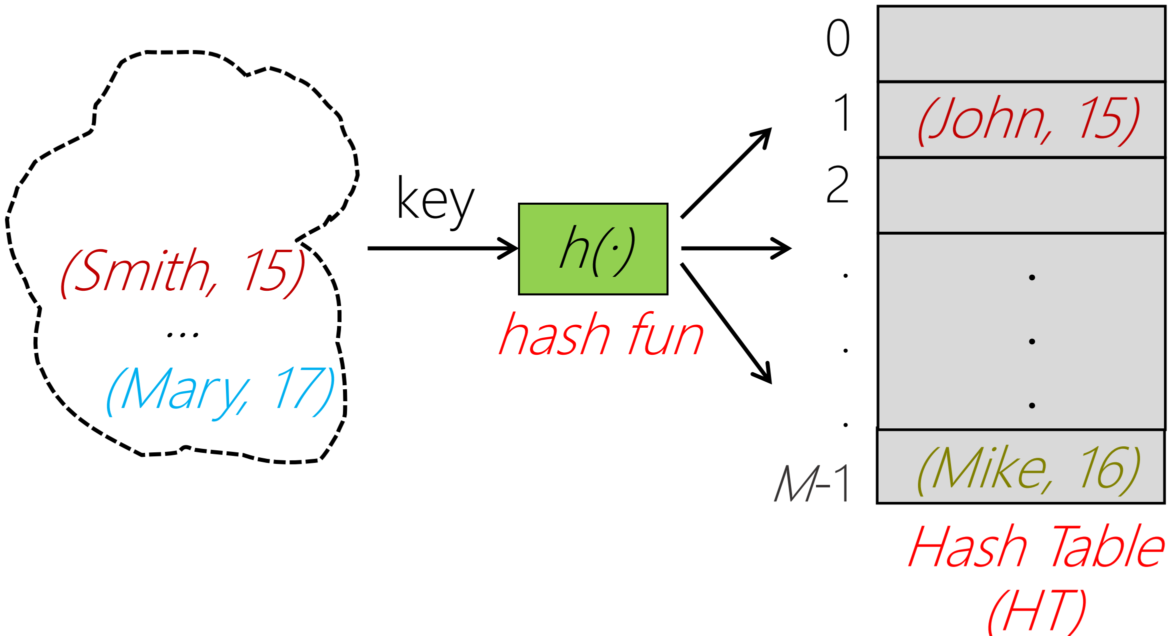
Hashing

- Mapping a key value to a position in a hash table (HT)
  - Hash table (array) *HT[0..M-1]*
    - Each position in *HT* is known as a slot
    - A slot can normally hold only one pair (*key, info*)
  - Hash function *h(·): a set of keys → [0..M-1]*



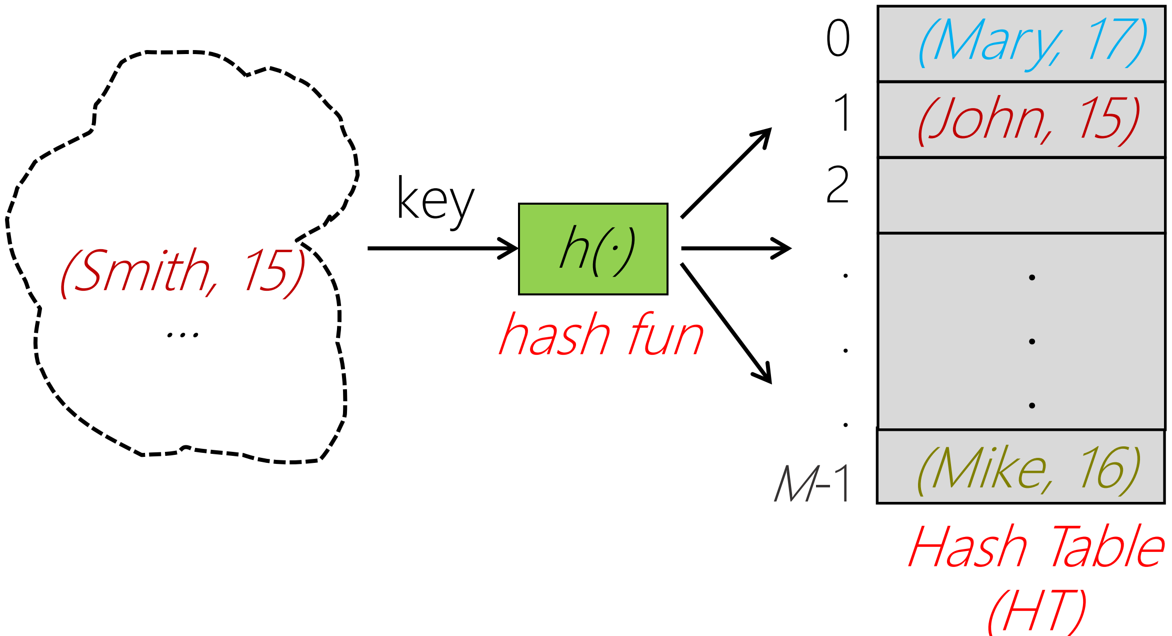
Hashing

- Mapping a key value to a position in a hash table (HT)
  - Hash table (array) *HT[0..M-1]*
    - Each position in *HT* is known as a slot
    - A slot can normally hold only one pair (*key*, *info*)
  - Hash function *h(·): a set of keys → [0..M-1]*



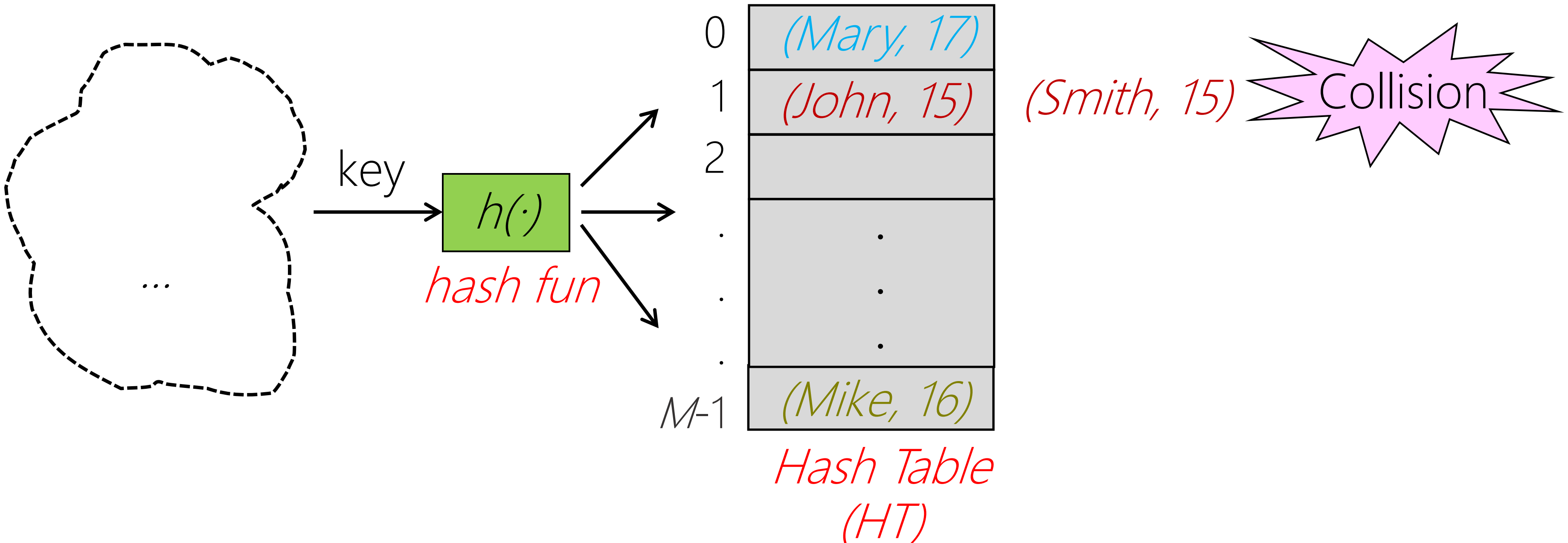
Hashing

- Mapping a key value to a position in a hash table (HT)
  - Hash table (array) *HT[0..M-1]*
    - Each position in *HT* is known as a slot
    - A slot can normally hold only one pair (*key*, *info*)
  - Hash function *h(·): a set of keys → [0..M-1]*



Hashing

- Mapping a key value to a position in a hash table (HT)
  - Hash table (array) $HT[0..M-1]$
    - Each position in HT is known as a slot
    - A slot can normally hold only one pair (*key*, *info*)
  - Hash function $h(\cdot)$: *a set of keys* $\rightarrow [0..M-1]$



Hashing Issues

- Choice of hash function
- Collision handling
  - Two different keys are mapped into the same location in HT
- Size (# of slots) of hash table
- Overflow handling
  - No space in the HT for a new pair (key, info)
  - When bucket size = 1, collisions & overflows occur simultaneously

Ideal Hash Functions

- What is an ideal hash function ?
  - Minimize the # of collisions
    - Distributes the keys *uniformly* over the slots of *HT*
    - A *random* key has an equal chance of hashing into any of the slots
  - Easy to compute

- Two parts of hash functions
  - Convert key into an integer (if it's not already an integer)
  - Map the integer into a slot address in *HT*

Designing Hash Functions

- Difficult to devise a good hash function
  - In general, the incoming keys are highly clustered (poorly distributed)

- Two situations when designing *HF*
  - When we know nothing about the distribution of the incoming keys
    - A hash function that generates a *uniform* random distribution

  - When we know something
    - A *distribution-dependent* hash function

Types of Hash Functions

- Division
  - Take the remainder of division by using modulus (%) operator
  - Keep the key values within the range of HT
- Folding
  - Partition the key value into several parts, then add the parts together
- Mid-square
  - Square the (integer) key value, and then take the middle r bits of the result (for a table of size 2^r)
- Digit analysis
  - When all the keys have been known in advance
  - Select certain digits of keys by deleting those digits that have the most skewed distribution (less useful to the uniform distribution), & then manipulate the rest digits)

Hashing by Division

- Example:

```
int hash(int x) {
    return(x % M);
}
```

  - M is the size of the hash table
  - The division remainder lies within the range of HT

- When key space = all integers, it's a uniform hash function

- In practice, keys tend to be correlated & clustered
  - Thus, the choice of *the divisor M* is critical to a good hash function

Choice of Divisor (*M*): Even vs. Odd

- When the divisor *M* is an *even* number
  - It cannot generate all possible hash values over HT
    - Odd (even) integers are hashed into odd (even) slots, respectively
  - Example:
    - 20%14 = 6, 30%14 = 2, 8%14 = 8
    - 15%14 = 1, 3%14 = 3, 23%14 = 9
  - We should NOT use an even divisor
- When the divisor *M* is an *odd* number
  - Odd (even) integers may be hashed into any slot
  - Example:
    - 20%15 = 5, 30%15 = 0, 8%15 = 8
    - 15%15 = 0, 3%15 = 3, 23%15 = 8
  - Better chance of uniformly distributed slots

Choice of Divisor (*M*): Prime Number

- In practice, an odd-number divisor may show similar biased distribution of HT slots
  - When the divisor is a multiple of prime numbers (such as 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, …)
  - (e.g.) M = 651: odd (= 3 * 7 * 31)
- But, the negative effect of each prime factor *p* of *M* decreases as *p* gets larger
- Ideally, choose large prime number *M*
- Alternatively, choose *M* so that it has *no prime factor* (< 20)
  - Example (integer factorization):
    - M = 651 (= 3 * 7 * 31): Bad
    - M = 713 (= 23 * 31): Good

Why Prime Number?

- If the input keys are uniformly-random distributed,
  - Prime divisor is not needed

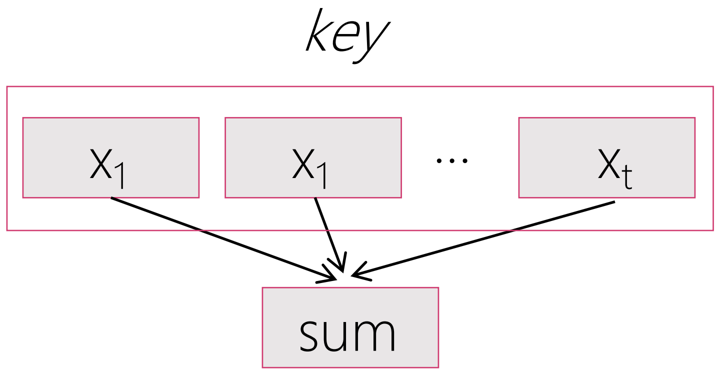
- What if input keys have some particular patterns
  - Keys: 10, 20, 30, 40, 50
  - Key mod 4 => 2, 0, 2, 0, 2 → bad
  - Key mod 7 => 3, 6, 2, 4, 1 → better

- Small vs large prime numbers
  - Key mod 5 => 0, 0, 0, 0, 0 → bad
  - Why? Multiple! So, let's use a large prime number

String Folding Method

- Example:

```
int hash(char* x) {
   int i, sum;
   for (sum=0, i=0; x[i]!='\0'; i++)
       sum += (int)x[i];
   return(sum % M);
}
```



- Breaking up the key value into several parts & combine them in some way

Folding Example

- Key
  - 123456789
- Hash table
  - 0~9
- Example hash function
  - Folded keys
    - 123, 456, 789
  - (123+456+789) mod 10

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

www.alpharithms.com

- Can be applied to characters (ASCII code)

Mid-Square Method

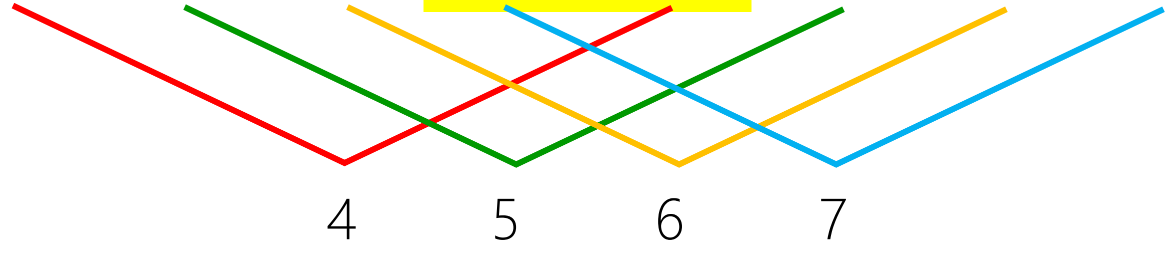
- A good hash function to use with integer key values
- Two steps:
  - Square the key value
  - Take out the middle r bits of the result (for a table of size 2^r), giving a value in the range 0 to 2^r -1

- Example (in decimal numbers):
  - Keys: 4-digit number
  - Goal: hashing into a table of size M = 100
    - The range of 0 to 99 is equivalent to two digits (i.e. $r = 2$)
  - If key = 4567 → squared value = 20857489 → hash value = 57

Why Mid-Square Good?

- All digits (4, 5, 6, 7) of the original key value contribute to the middle two digits (5, 7) of the squared value

				4	5	6	7
				4	5	6	7
			3	1	9	6	9
		2	7	4	0	2	
	2	2	8	3	5		
1	8	2	6	8			
2	0	8	5	7	4	8	9



Types of Hashing

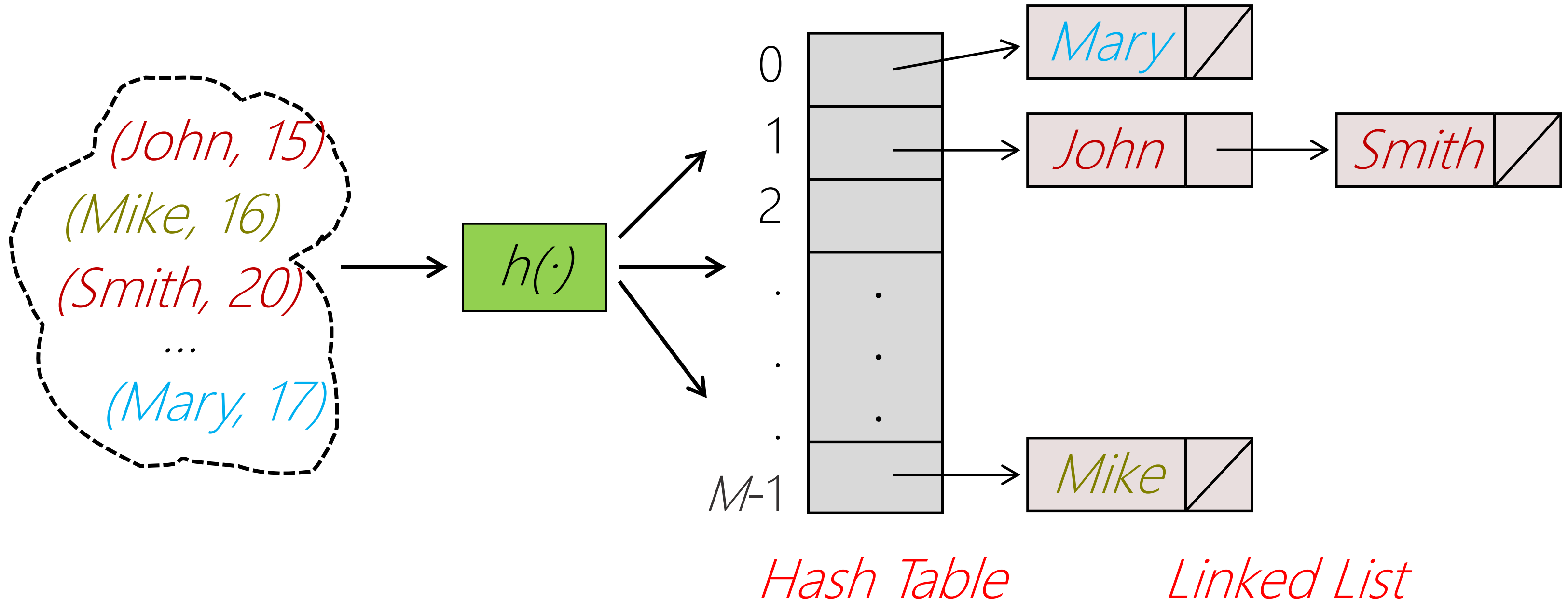
- Static hashing
  - Hash function (& thus HT size) is fixed
  - How to handle 'collision/overflow':
    - *Open* hashing (separate chaining)
    - Closed hashing (*open* addressing)
  - Unacceptable performance as data grows with time
    - Need to reorganize the hash structure - expensive

Don't be confused

- Dynamic hashing (NOT Covered)
  - Hash function (& HT size) is allowed to be modified dynamically
    - Extensible hashing
    - Linear hashing
  - Good for DBMS (DataBase Management System) that grow & shrink in size

Open Hashing

- Also called separate chaining



- Appropriate when
  - *HT* is kept in main memory with *in-memory* linked list
  - Avoid multiple disk accesses

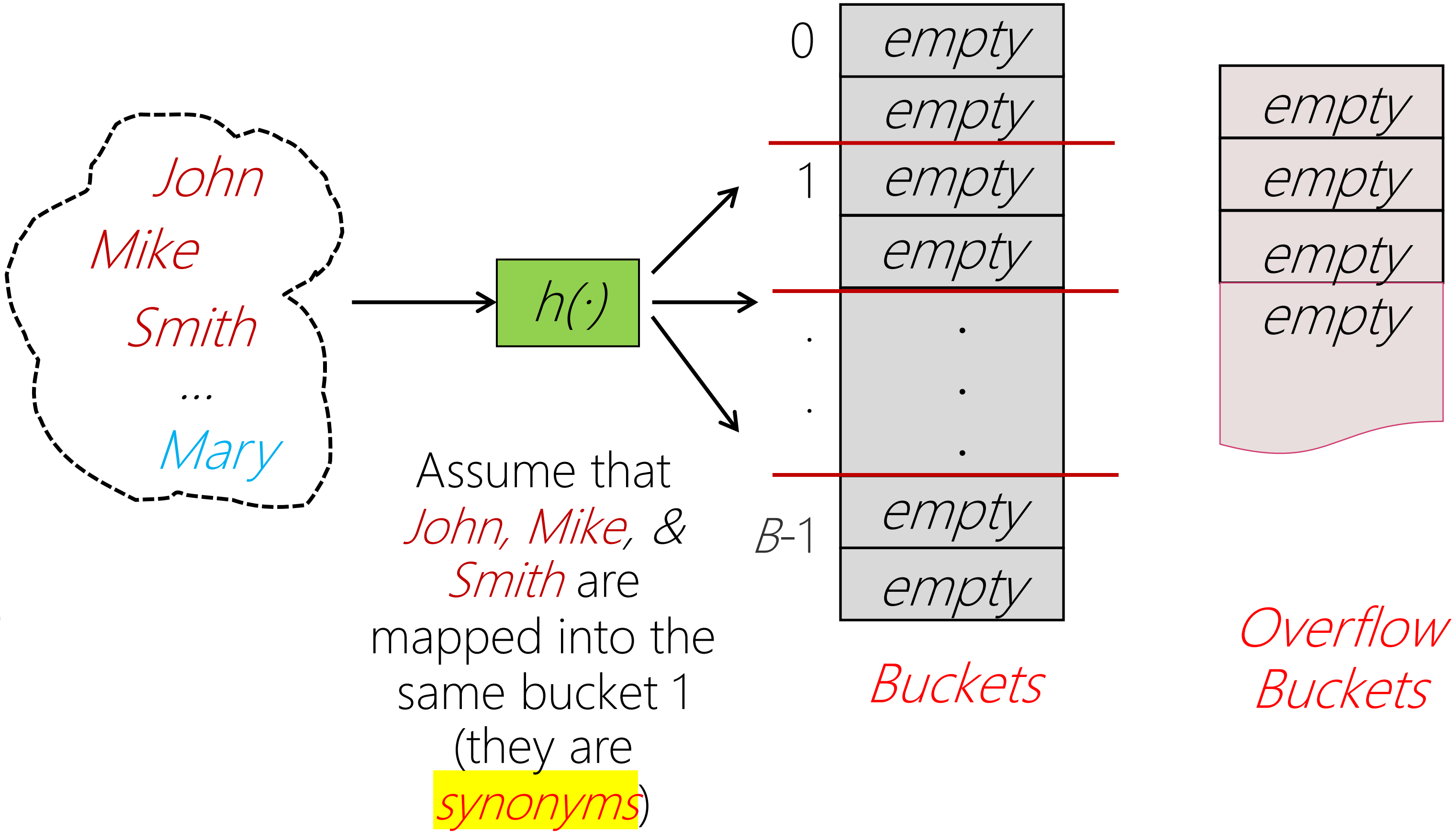
Closed Hashing

- Stores all elements directly in hash table
  - Each slot of HT is marked by one of three states
    - *empty*, *occupied*, or *deleted* (why?)

- Two type of implementations:
  - Bucket hashing
    - HT slots are grouped into buckets
    - Overflow bucket of infinite capacity
      - Shared by all buckets
  - Rehashing
    - No bucketing
    - Probing (also called *open addressing*)

Bucket Hashing

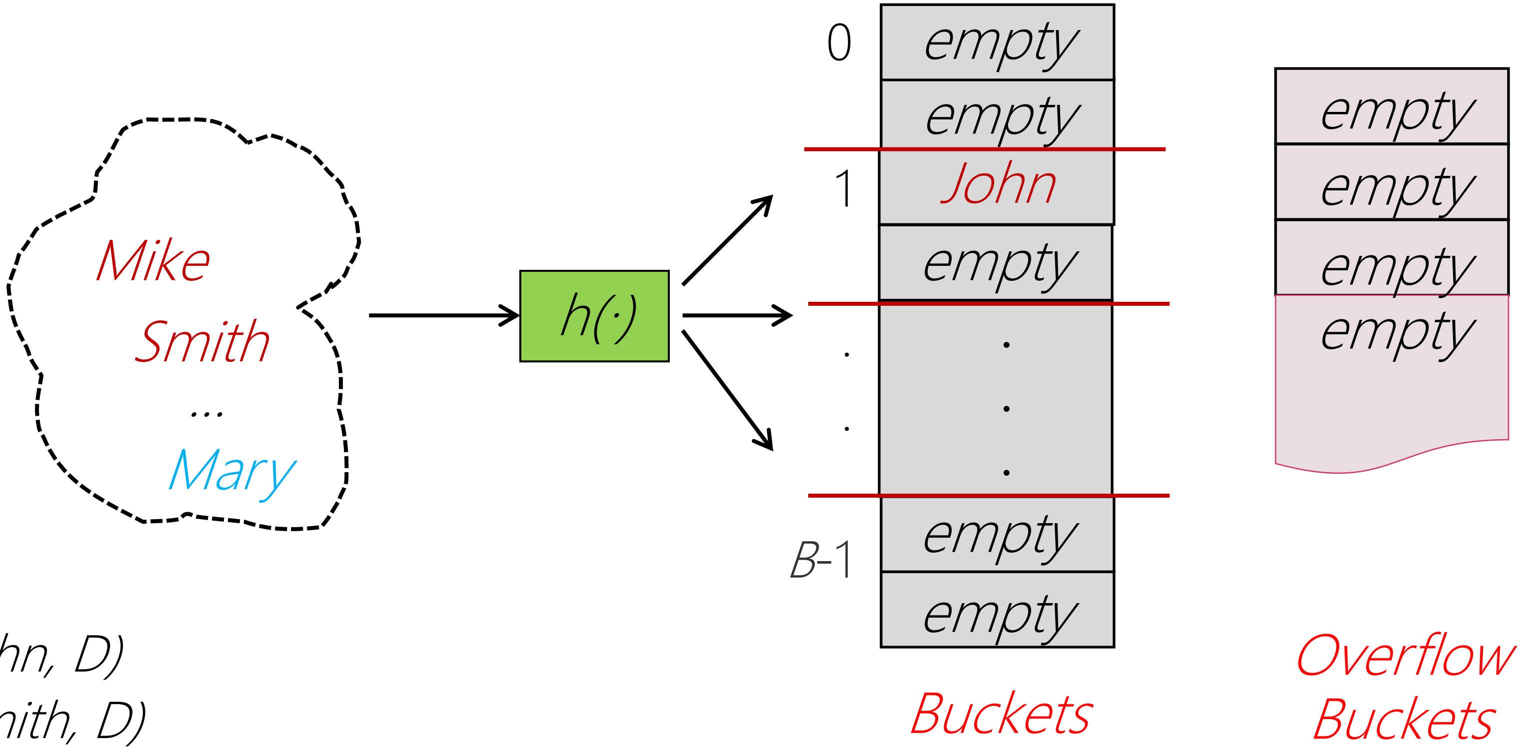
- HT slots are grouped into buckets



- *Insert(John, D)*

Bucket Hashing

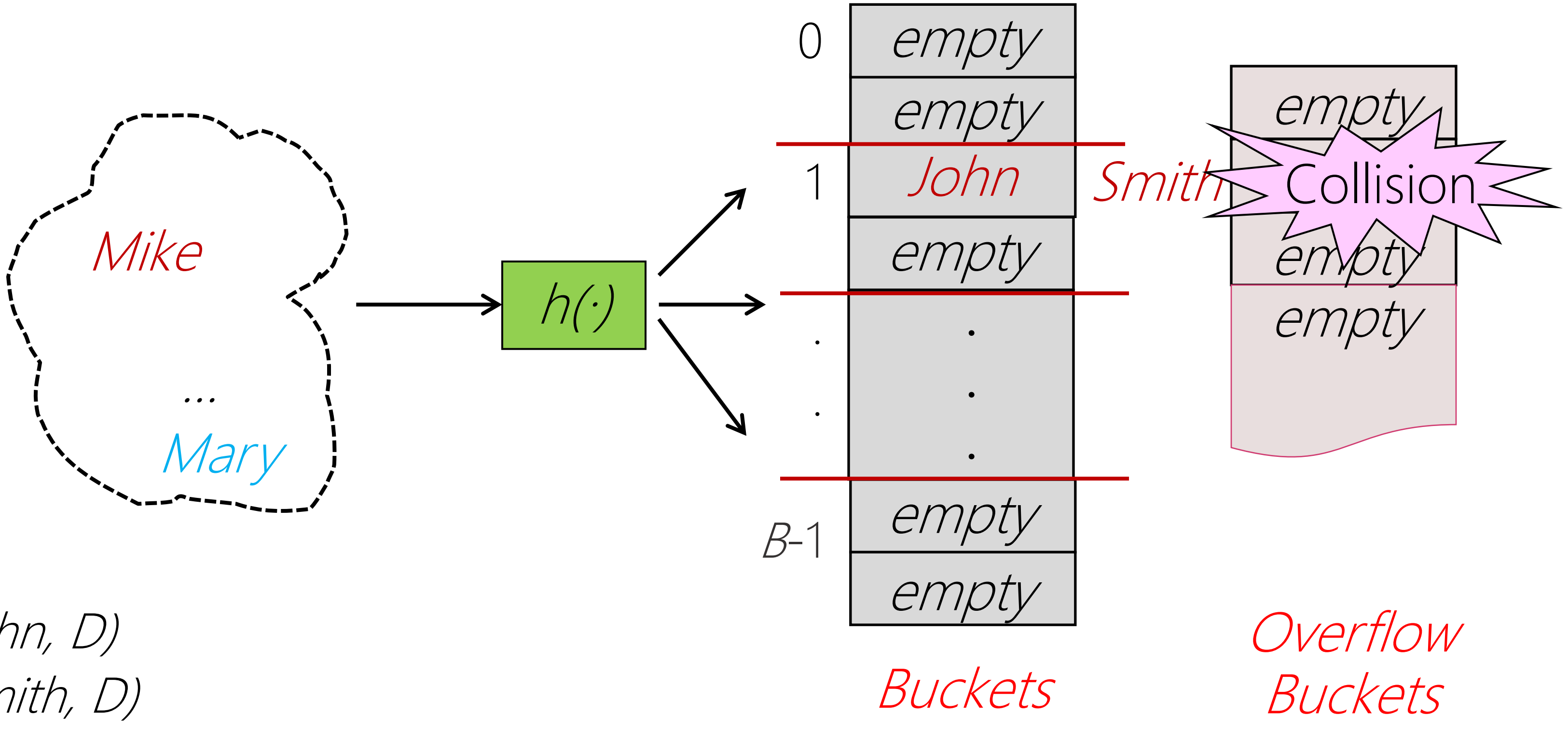
- HT slots are grouped into buckets



- After *Insert(John, D)*
- Then *Insert(Smith, D)*

Bucket Hashing

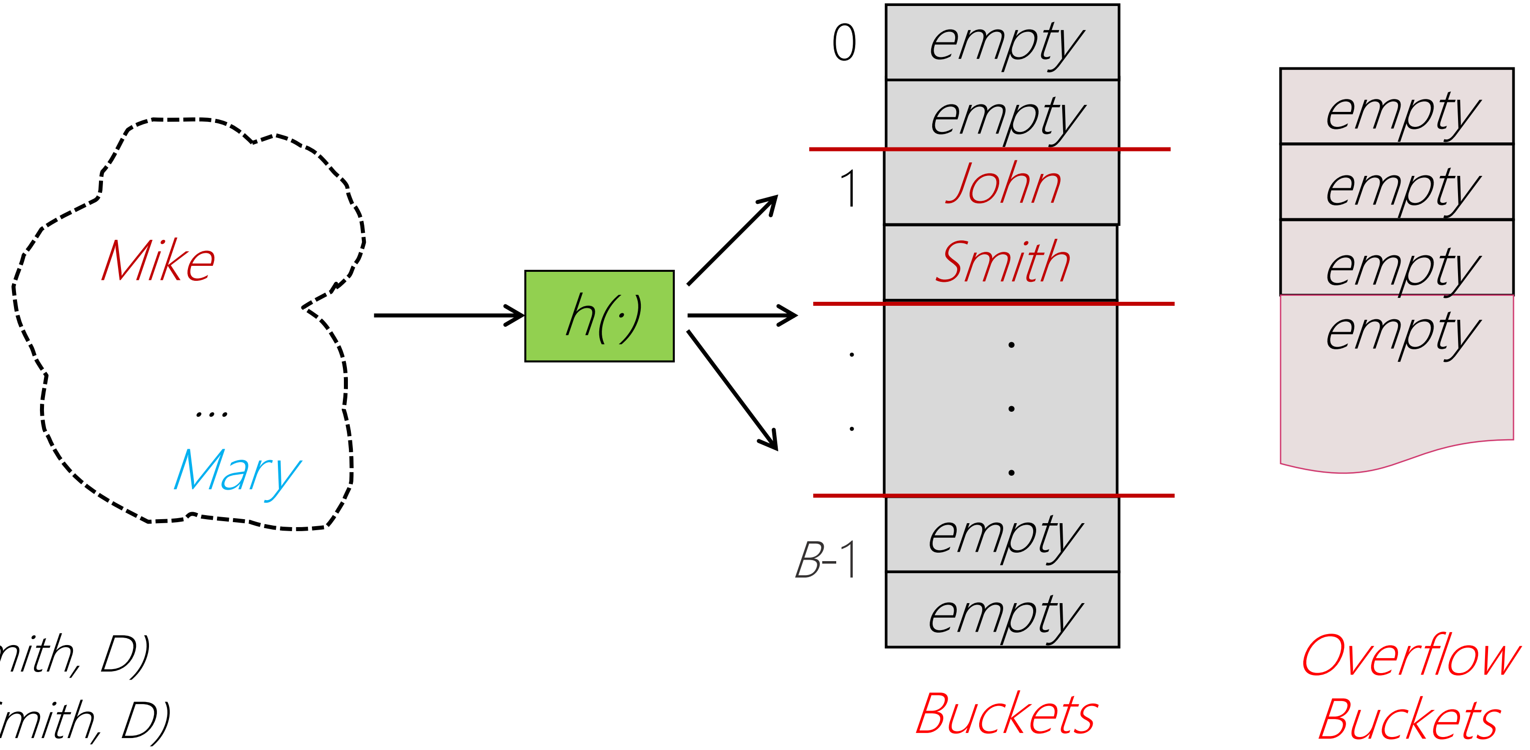
- HT slots are grouped into buckets



- After *Insert(John, D)*
- Then *Insert(Smith, D)*

Bucket Hashing

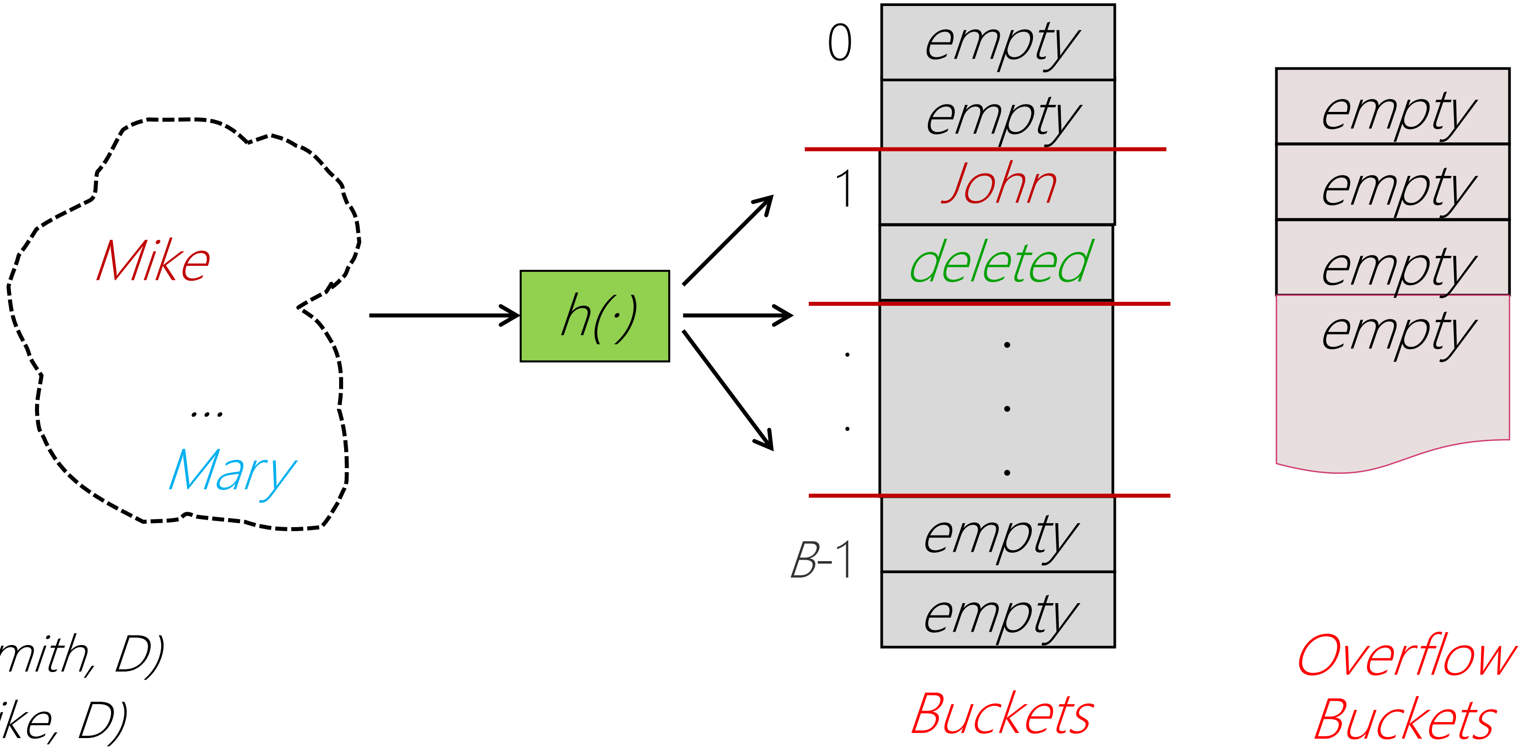
- HT slots are grouped into buckets



- After *Insert(Smith, D)*
- Then *Delete(Smith, D)*

Bucket Hashing

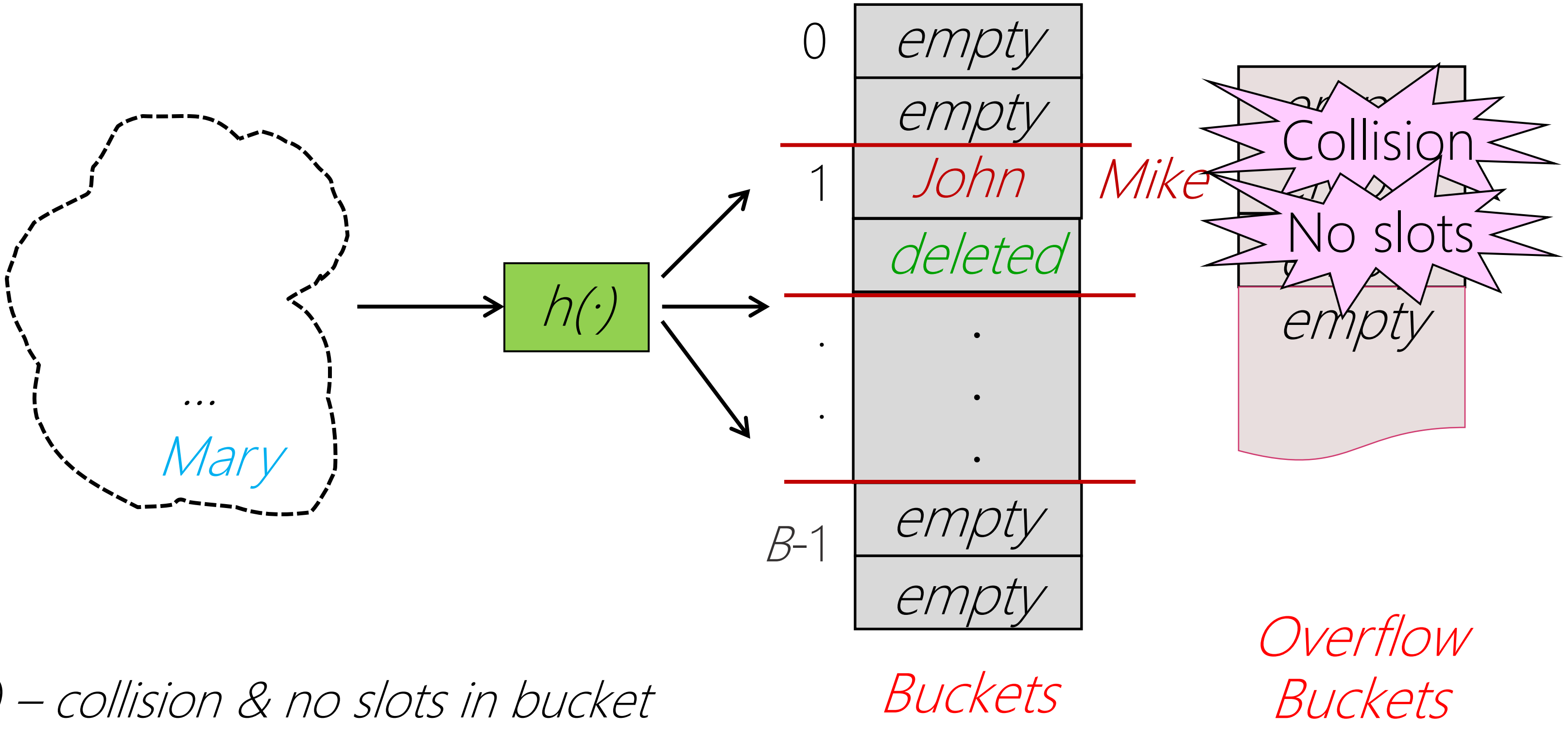
- HT slots are grouped into buckets



- After *Delete(Smith, D)*
- Then *Insert(Mike, D)*

Bucket Hashing

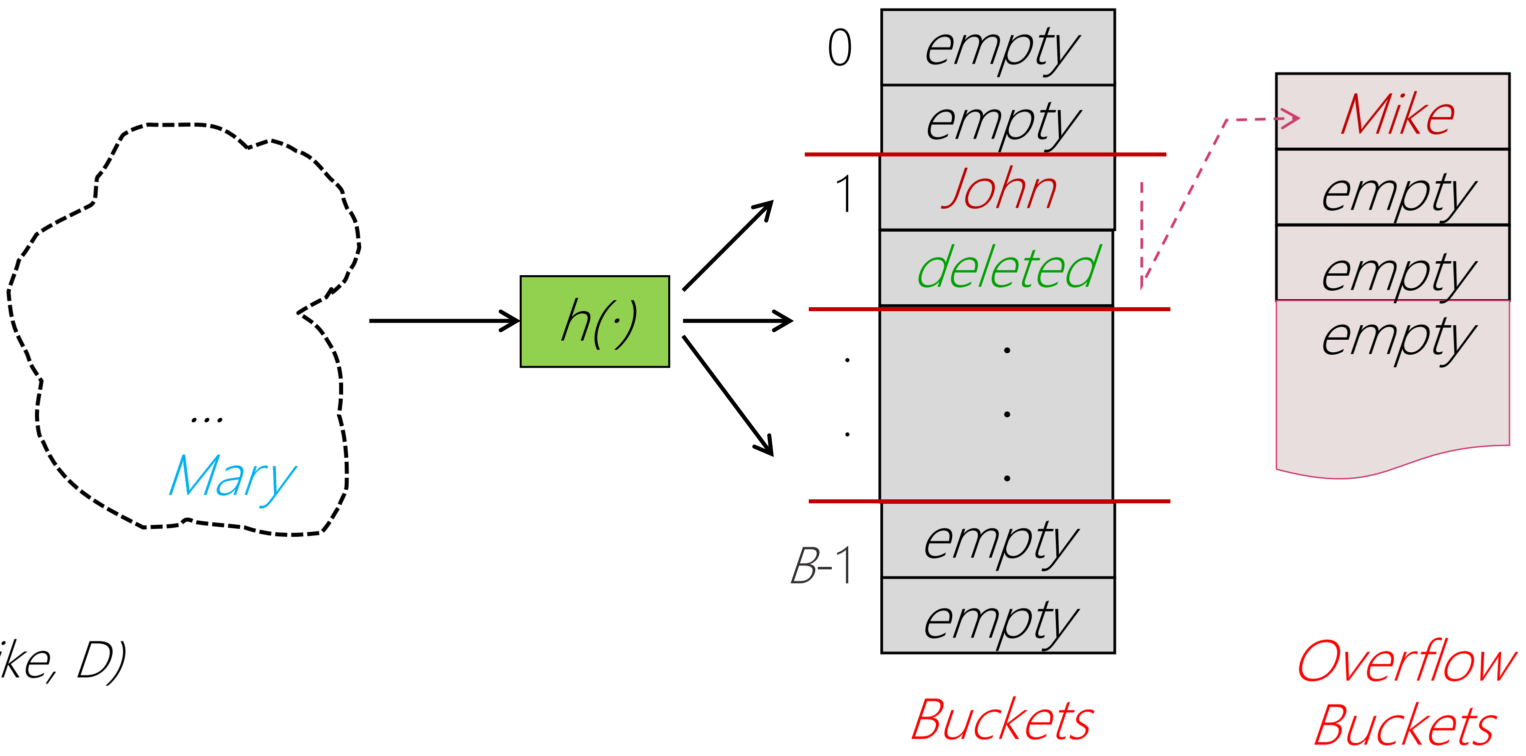
- HT slots are grouped into buckets



- *Insert(Mike, D) – collision & no slots in bucket*

Bucket Hashing

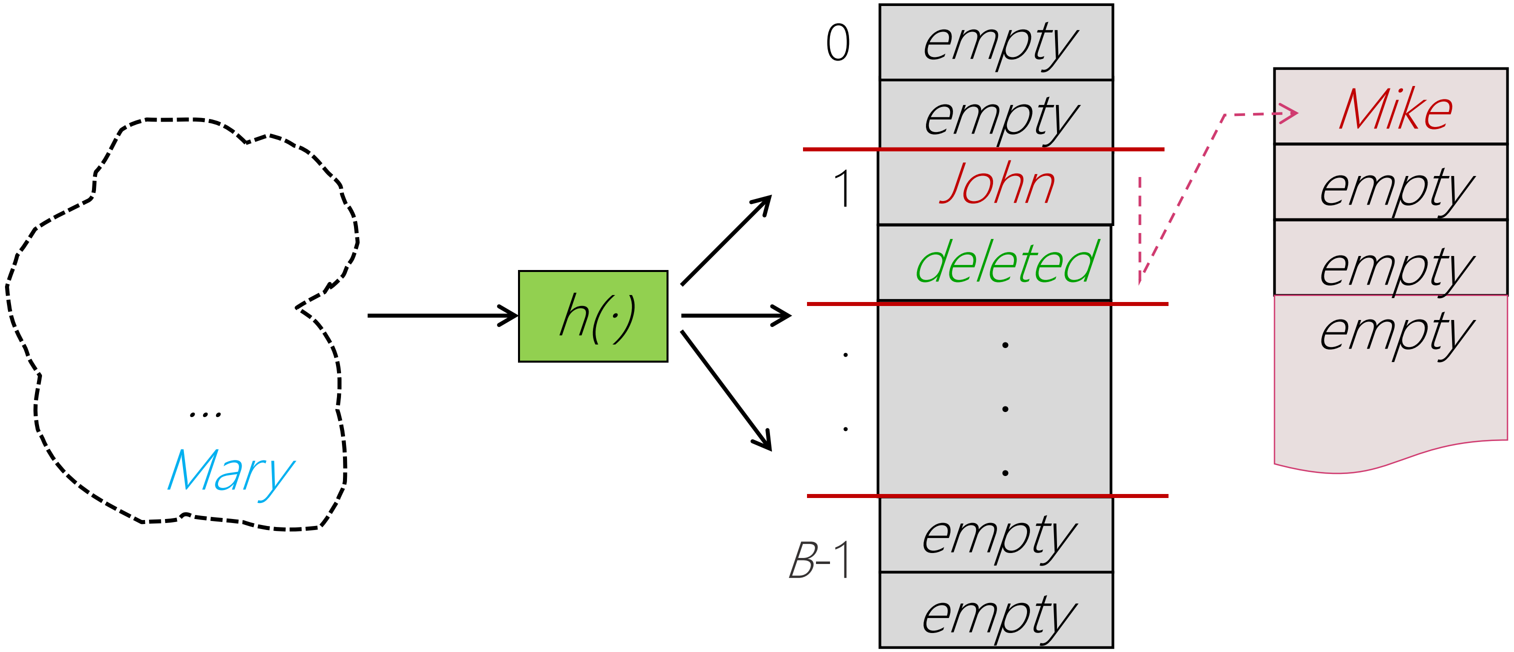
- HT slots are grouped into buckets



- After *Insert(Mike, D)*

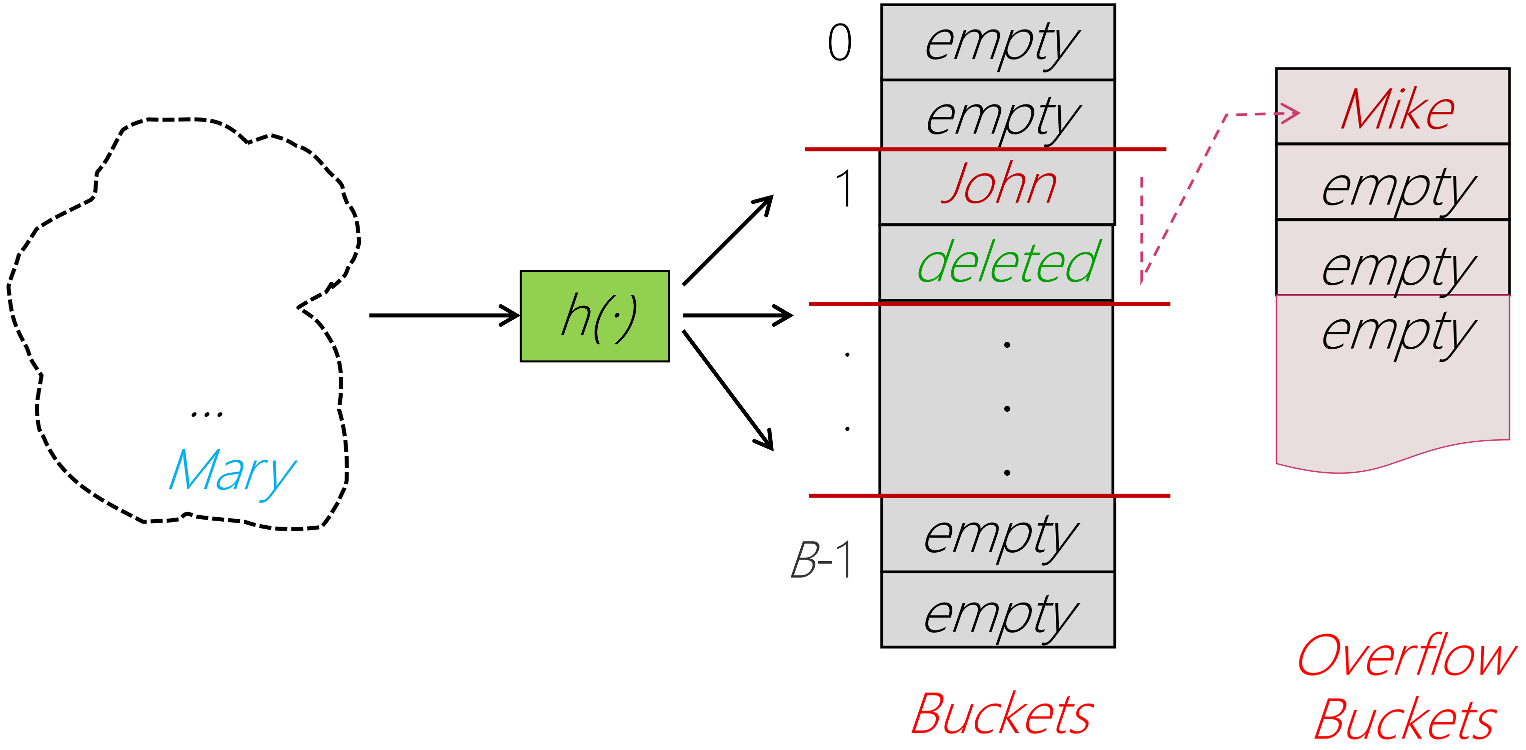
Bucket Hashing

- HT slots are grouped into buckets



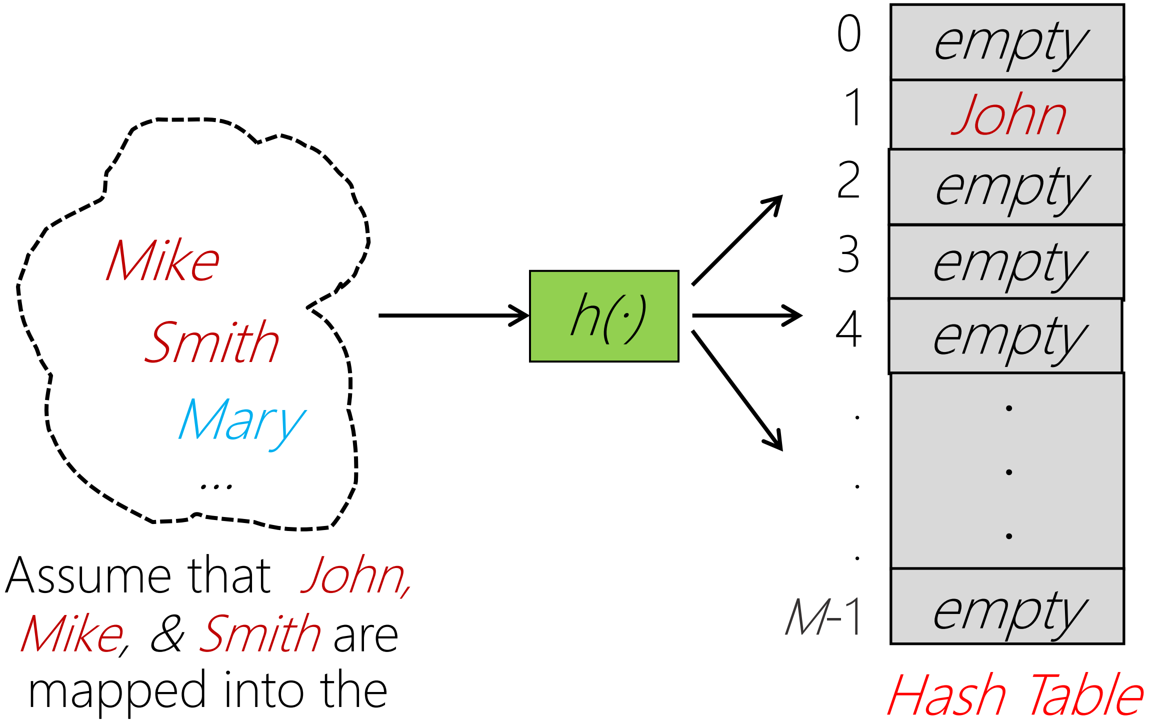
- Quiz:
  - Can we use the "*deleted*" slot to store a new record?

Bucket Hashing



- Good for implementing HTs stored on *disk*
  - Bucket size can be set to the size of disk block

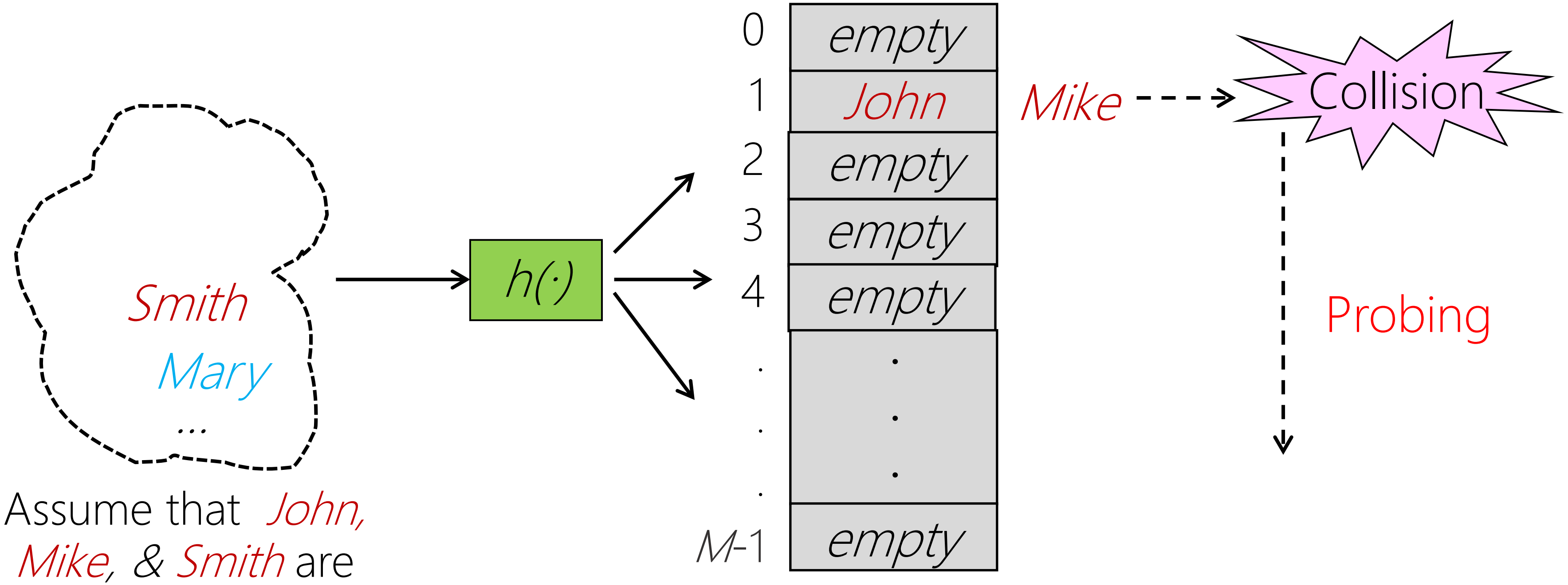
Rehashing (Open Addressing)



Assume that *John*, *Mike*, & *Smith* are mapped into the same slot 1

- When *Insert* (*Mike*, *D*)

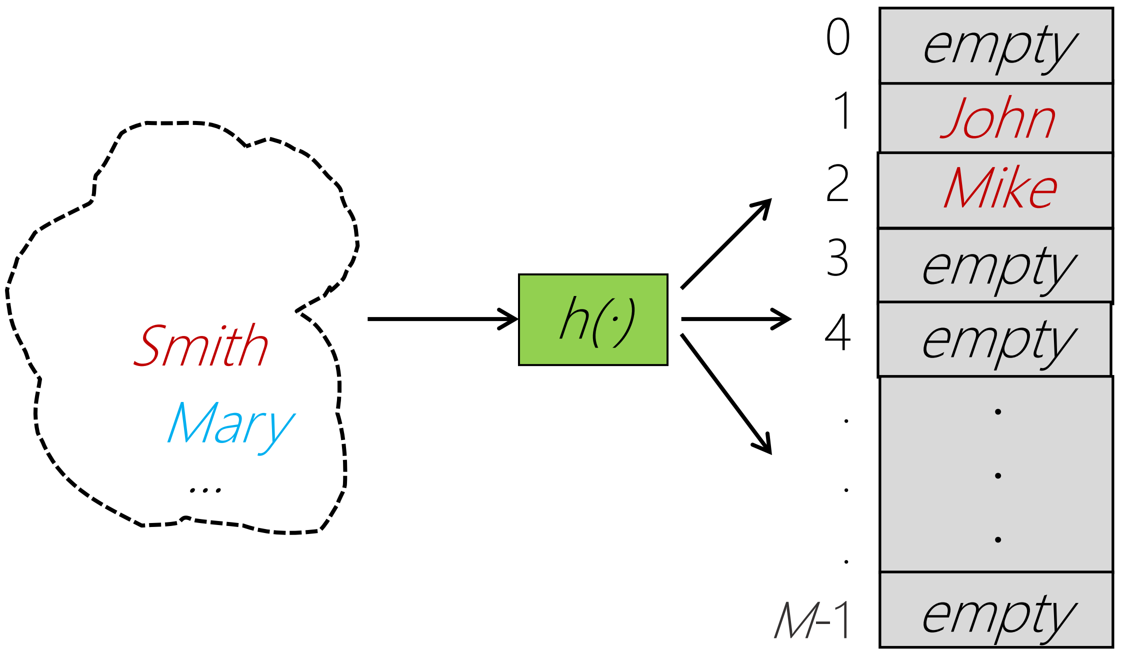
Rehashing (Open Addressing)



Assume that *John*, *Mike*, & *Smith* are mapped into the same slot 1

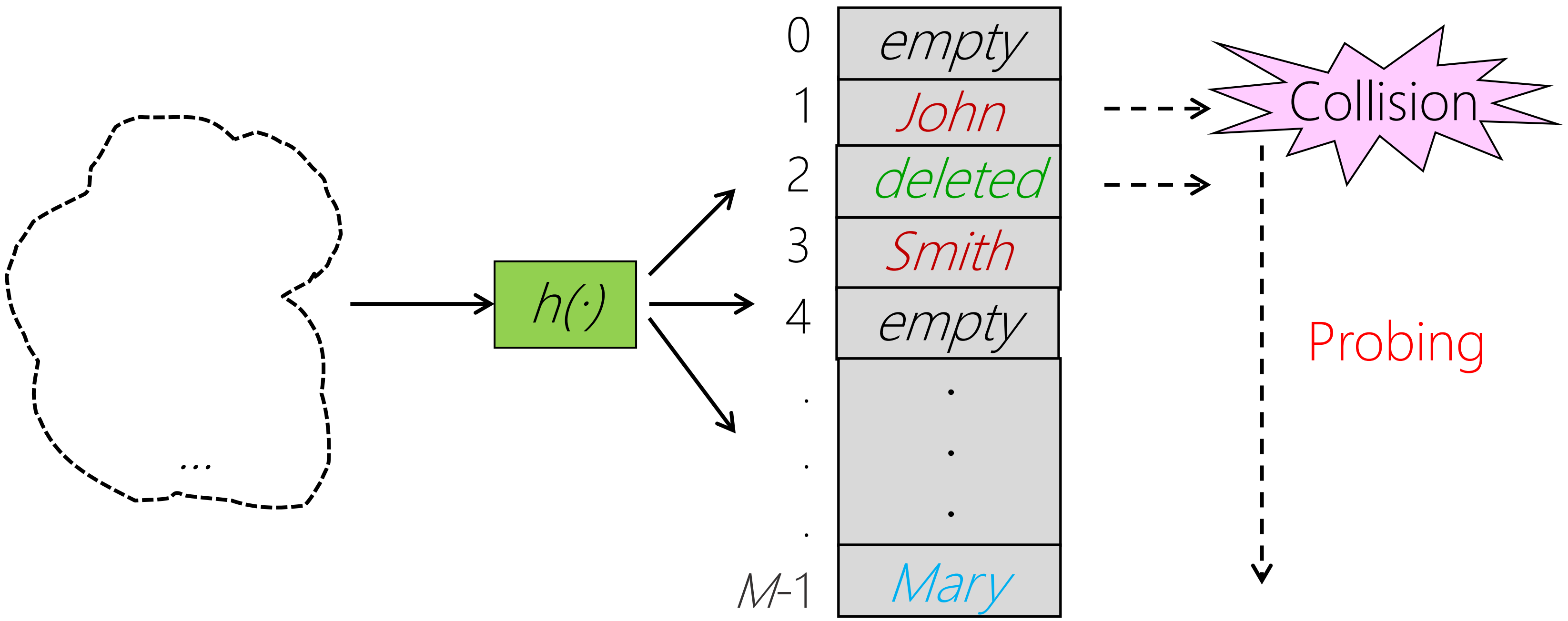
- When *Insert* (*Mike*, *D*)
  - A collision occurs, alternative slots are tried by *rehashing*
  - Next available slot? : $h_i(x) = (h(x) + i\ ) \bmod M$

Rehashing (Open Addressing)



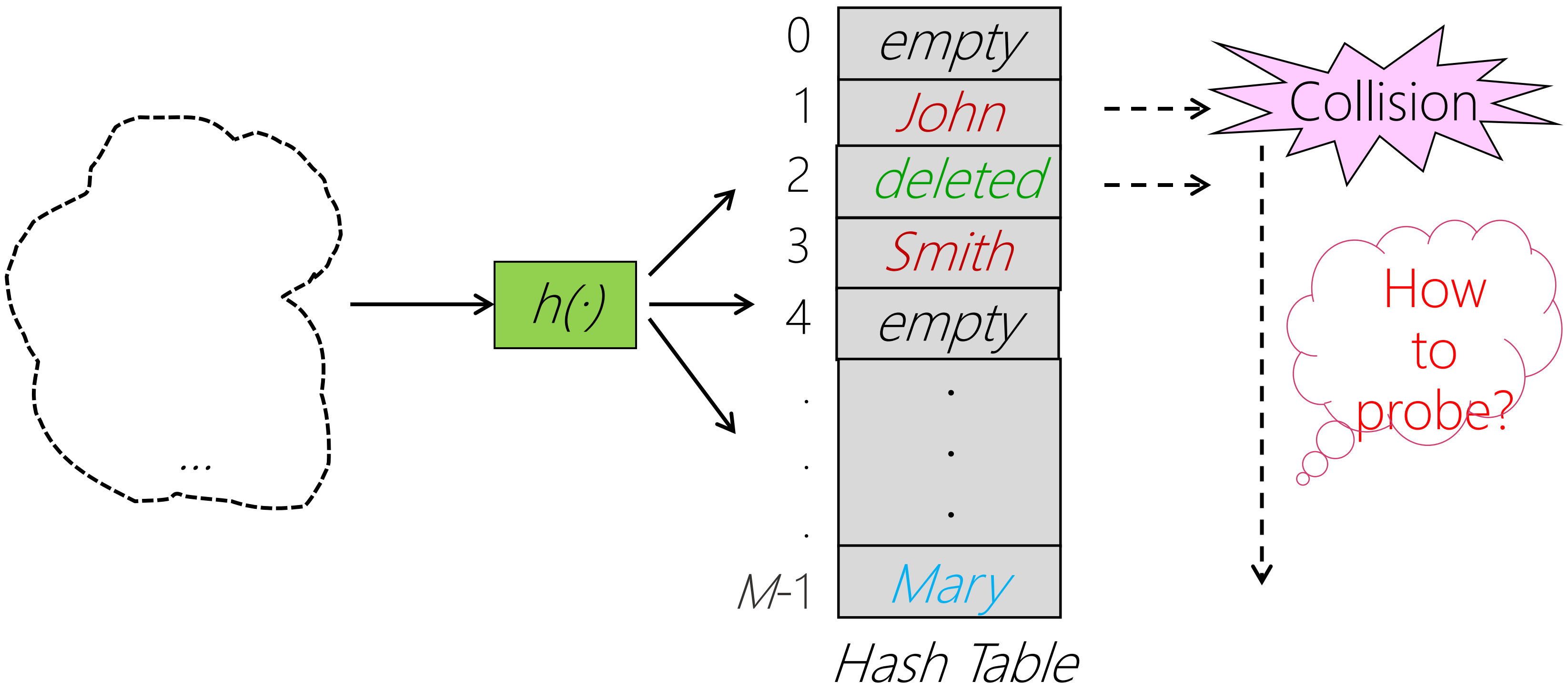
- After *Insert* (*Mike, D*)
- Then *Insert*(*Smith, D*), *Delete*(*Mike, D*), & *Insert*(*Mary, D*)

Rehashing (Open Addressing)



- After *Insert*(*Smith*, *D*), *Delete*(*Mike*, *D*), & *Insert*(*Mary*, *D*)

Rehashing (Open Addressing)



- Rehashing & probe function

$$h_i(x) = (h(x) + p(i)) \bmod M, \quad \text{where } p(i): \text{ probe function}$$

Linear Probing

- Probe function
  - Typically $p(i) = i$, thus $p(i)$: a linear function of i

$$h_i(x) = (h(x) + i) \bmod M$$

- Definite drawback of linear probing
  - Primary Clustering
    - Tendency to cluster items together
    - Keys share substantial segments of a probe sequence
  - Leads to long probe sequence

Linear Probing: By Steps

- Linear probing, but skipping slots by a constant $c > 1$

$$h_i(x) = (h(x) + c \times i) \bmod M$$

  - Constant c must be relatively prime to M (that is, c and M must share no factors)
    - To visit all slots in HT before returning to the home position

- But, cluster still remains
  - Consider the situation where $c = 2$
    - $h(k_1) = 3$ → probe sequence = 3, 5, 7, 9, ...
    - $h(k_2) = 5$ → probe sequence = 5, 7, 9, ...
  - The probe sequences of k_1 & k_2 are linked together in a manner that contributes to clustering

How to Avoid Primary Clustering

- How to solve the problem of primary clustering?

- Can we probe HT *at random*?
  - Yes/No?:
  - Why?:

- Two popular ways:
  - Pseudo-random probing
  - Quadratic probing

Pseudo-Random Probing

- The i-th slot in the probe sequence is

$$h_i(x) = (h(x) + d_i) \bmod M$$

where $d_1, d_2, \ldots, d_{M-1}$: a random permutation of integers 1, 2, …, M-1

  - All insertions & searches must use the same sequence of random numbers

- One effective way of generating a random permutation
  - Using "*shift-register sequence*"

Shift-Register Sequence

Given M (a power of 2) and a constant k $(1 \le k \le M\text{-}1)$

Start with some number d_1 such that $1 \le d_1 \le M - 1$

Repeat to generate successive numbers $d_2, d_3, d_4, \ldots$

- Double the previous number
- If the result ≥ M, then
  - Subtract M and
  - Take the "bitwise modulo-2 sum" of
    - the result &
    - the selected constant k

(** The "bitwise modulo-2 sum" is a binary addition with carries ignored)

Example: Shift-Register Sequence

- Let $M = 8$, $k = 3$
- Start with d1 = (101) = 5
  - (1) Shift (Double): (1010) ≥M
    Delete leading 1 (= Subtract M): (010)
    ⊕ 3: d2 = (001) = 1
  - (2) Shift: d3 = (010) = 2
  - (3) Shift: d4 = (100) = 4
  - (4) Shift: (1000) ≥M
    Delete leading numbers: (000)
    ⊕ 3: d5 = (011) = 3

XOR operation (denoted as ⊕)

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

- Note:
  - Not every value of *k* will produce a permutation of 1, 2, …, *M-1*
  - However, for a given *M*, there are some *k* that works

Quadratic Probing

- The i-th slot in the probe sequence is

$$h_i(x) = (h(x) + p(i)) \bmod M$$

$$\text{where } p(i) = c_1 i^2 + c_2 i + c_3$$

- Simplest one

$$h_i(x) = (h(x) + i^2) \bmod M$$

Secondary Clustering

- Primary clustering can be eliminated by both *pseudo-random* & *quadratic* probing
- But, clusters still remain (Secondary clustering)
  - If two keys hash to the same home position, then they will always follow the same probe sequence
  - Why?
    - the probe sequence is entirely a function of the *home position*, NOT *the original key value*
  - Likely to cause a cluster to a particular position

Double Hashing

- To avoid secondary clustering
  - Probe function:

$$p(i) = i \times h_2(x)$$

where $h_2(x)$: a second hash function

References

- Further reading list and references
  - https://www.geeksforgeeks.org/folding-method-in-hashing/

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee