Hashing to Store Symbol Tables

Basics Collision resolution Universal hashing

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

World Airports

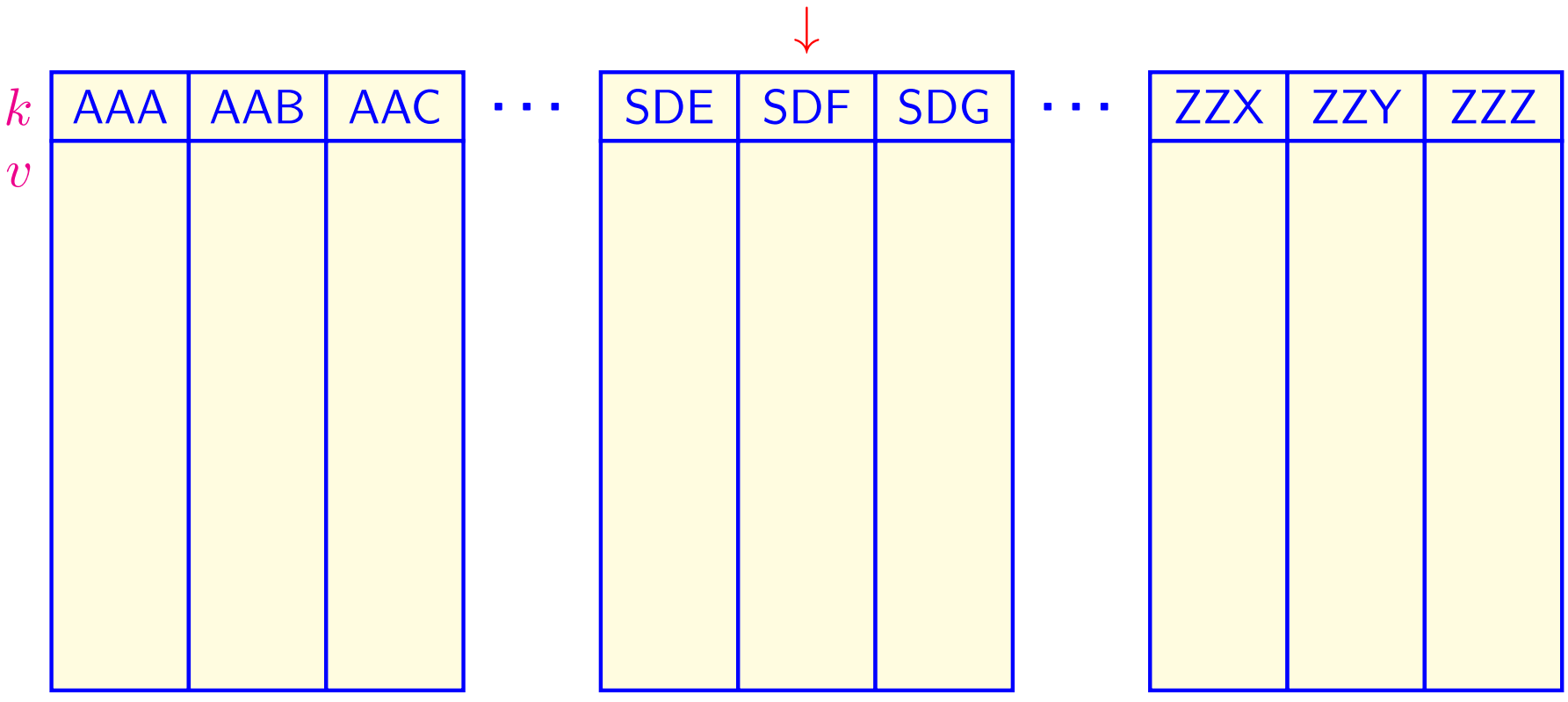
Textbook, Table 3.1 of airports ordered by symbolic codes:

Key, k	**Associated value**, v		
Code	City	Country	Place
AKL	Auckland	New Zealand	
DCA	Washington	USA	District Columbia
FRA	Frankfurt a.M.	Germany	Rheinland-Pfalz
GLA	Glasgow	UK	Scotland
HKG	Hong Kong	China	
LAX	Los Angeles	USA	California
ORY	Paris-Orly	France	
SDF	Louisville	USA	Kentucky

Array implementation: works well, provided the number of possible search keys is sufficiently small (here, $26^3 = 17,576$ codes).

Table of Airports: Array Implementation

Search by direct addressing, e.g., $k^\circ = \mathsf{SDF}$



Symbol Table and Hashing

Usually, only a tiny fraction of all possible keys are actually in use:

UoA student ID: 7-digit / 9-digit decimal number

10,000,000 possible keys $\Leftrightarrow \approx 40,000$ students (0.4%) annually.
1,000,000,000 possible keys $\Leftrightarrow \approx 40,000$ students (0.004%).

- (Symbol) **table** is a set of table entries, (k, v), such that each entry contains a unique key, k, and a value (information), v.
- Each key uniquely identifies its entry.
- Table searching:
  - **Given:** a search key, k
  - **Find:** the table entry, (k, v)

Hashing to store and search for values in a symbol table uses less space than *direct array addressing*, but retains many of its benefits.

Symbol Table and Hashing

Hashing stores values and searches for them in:

- Linear, $O(n)$, worst-case time and
- Extremely fast, $O(1)$, average-case time.

Once the entry (k, v) of a symbol table is found:

- its value v, may be updated, or
- it may be retrieved, or
- the entire entry, (k, v), may be removed from the table.

If no entry with key k exists in the table:

- A new entry with k as its key may be inserted to the table.

Basic Features of Hashing

Hashing computes an integer **hash code**, for each object.

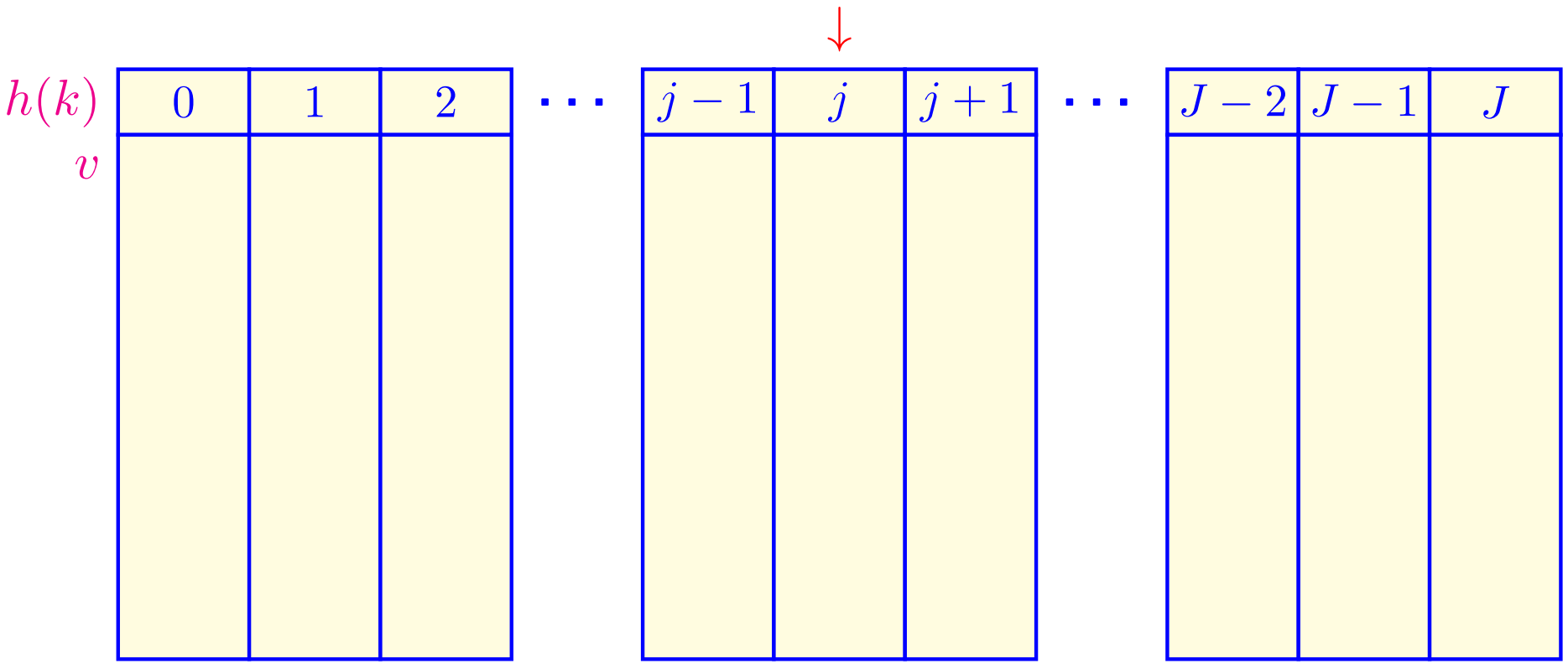
- The computation implements a **hash function,** $h(k)$.
- It maps objects (e.g., keys k) to indices of a given linear array, called the **hash table**.
- The function must always return a valid array index.

An object with a key k has to be stored at the location $h(k)$.

- Hash codes must be computed quickly.
- Hashing a key to an index depends only on the key to hash.
- It is independent of all other keys in the table.

Hash Table

Search for an object with a key k° at the location $j = h(k^\circ)$



Basic Features of Hashing

Perfect hash function:

- a different index value for every key. But such a function cannot be always found.

Collision:

- if two distinct keys, $k_1 \neq k_2$, hash to the same hash address, $h(k_1) = h(k_2)$

Collision resolution policy:

- how to find additional storage to store one of the collided table entries

Load factor λ:

- fraction of the already occupied entries (n occupied entries in the table of size m: $\lambda = \frac{n}{m}$)

Example 3.24 (Textbook): Mapping Keys to Array Indices

- 100 two-digit integer keys k: 00,01,...,99
- 10 array indices: 0,1,...,9
- A simple rounding hash function: $h(k) = \lfloor \frac{k}{10} \rfloor$ returning the closest integer, which is less than or equal to $\frac{k}{10}$, e.g.,

$$h(3) = \left\lfloor \frac{3}{10} \right\rfloor = \lfloor 0.3 \rfloor = 0; \quad h(77) = \left\lfloor \frac{77}{10} \right\rfloor = \lfloor 7.7 \rfloor = 7; \quad \ldots$$

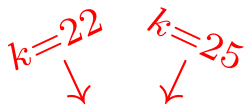
$h(k)$	0	1	2	3	4	5	6	7	8	9
v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9

Example 3.24 (Textbook): Mapping Keys to Array Indices

Collision: e.g., the keys 22 and 25 hash to the same index, $h(22) = h(25) = 2$:

$$h(22) = \left\lfloor \frac{22}{10} \right\rfloor = \lfloor 2.2 \rfloor = 2; \quad h(25) = \left\lfloor \frac{25}{10} \right\rfloor = \lfloor 2.5 \rfloor = 2; \quad \ldots$$

Collision resolution policy is to be defined to store both the table entries, (k_1, v_1) and (k_2, v_2).



$h(k)$	0	1	2	3	4	5	6	7	8	9
v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9

- Different keys hashed to the same hash address are called **synonyms**.
- Data items with synonymic keys are often called *synonyms*, too.

How Common Are Collisions?

Richard von Mises [1883–1953]: von Mises Birthday Paradox:

if there are more than 23 people in a room, the chance is greater than 50% (!) that two or more of them have the same birthday [a].

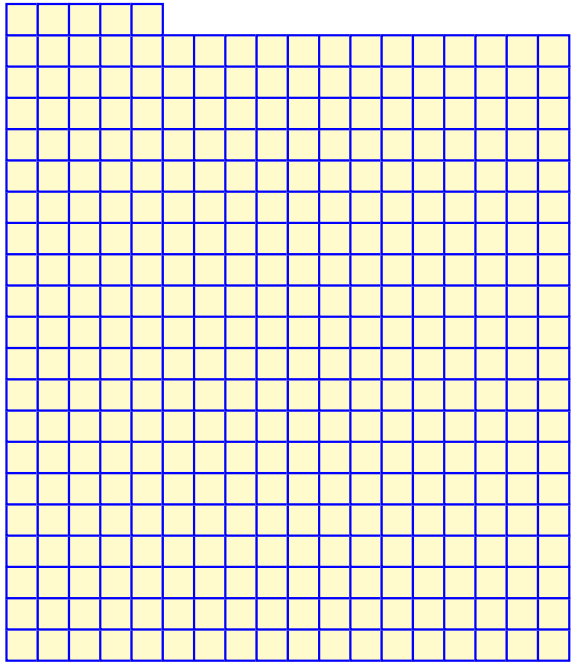
[a] It is not really a paradox in the math sense, but a counter-intuitive fact!

If the table is only 6.3% full (since $\frac{23}{365} = 0.063$), the chance of a collision is better than 50%!

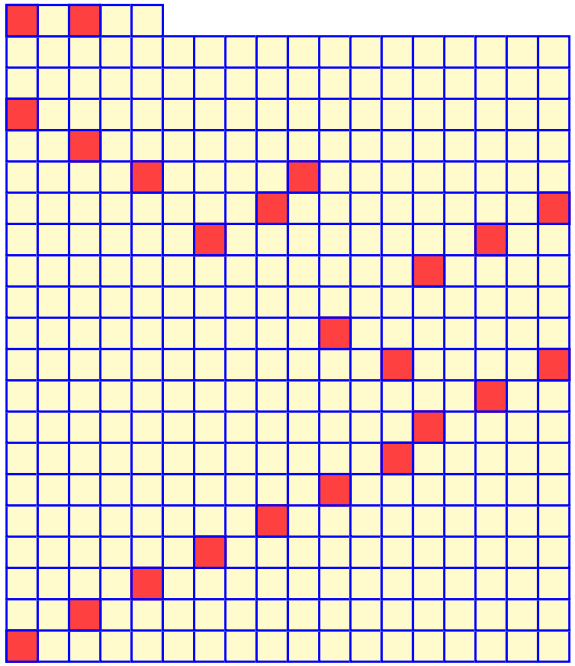
- 50.0% chance of collision if $\lambda = 0.063$ (23 out of 365).
- 89.1% chance of collision if $\lambda = 0.110$ (40 out of 365).
- 99.4% chance of collision if $\lambda = 0.164$ (60 out of 365).

How Common Are Collisions?

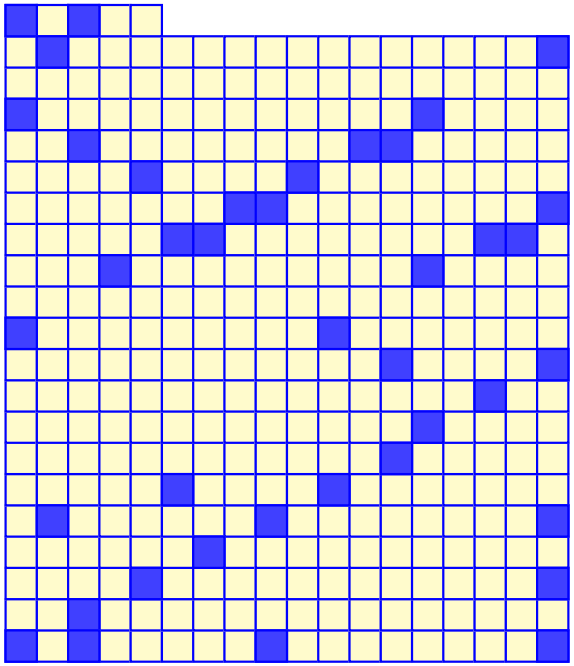
$m = 365$



$\lambda = \frac{0}{365} = 0$

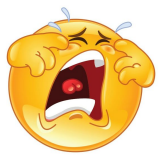


$\frac{23}{365} = 0.063$



$\frac{40}{365} = 0.110$

Chance of collision: 0% 50.7% 89.1%



How Common Are Collisions?

Probability $Q(m,n)$ of no collision between n items, being randomly tossed into a table with m slots:

$$Q(m,1) = 1 \equiv \frac{m}{m} \qquad m \text{ free slots}$$

$$Q(m,2) = Q(m,1)\frac{m-1}{m} \equiv \frac{m(m-1)}{m^2} \qquad m-1 \text{ free slots}$$

$$Q(m,3) = Q(m,2)\frac{m-2}{m} \equiv \frac{m(m-1)(m-2)}{m^3} \qquad m-2 \text{ free slots}$$

$$\ldots \qquad \ldots$$

$$Q(m,n) = Q(m,n-1)\frac{m-n+1}{m} \qquad m-n+1 \text{ free slots}$$

$$\equiv \frac{m(m-1)\cdots(m-n+1)}{m^n}$$

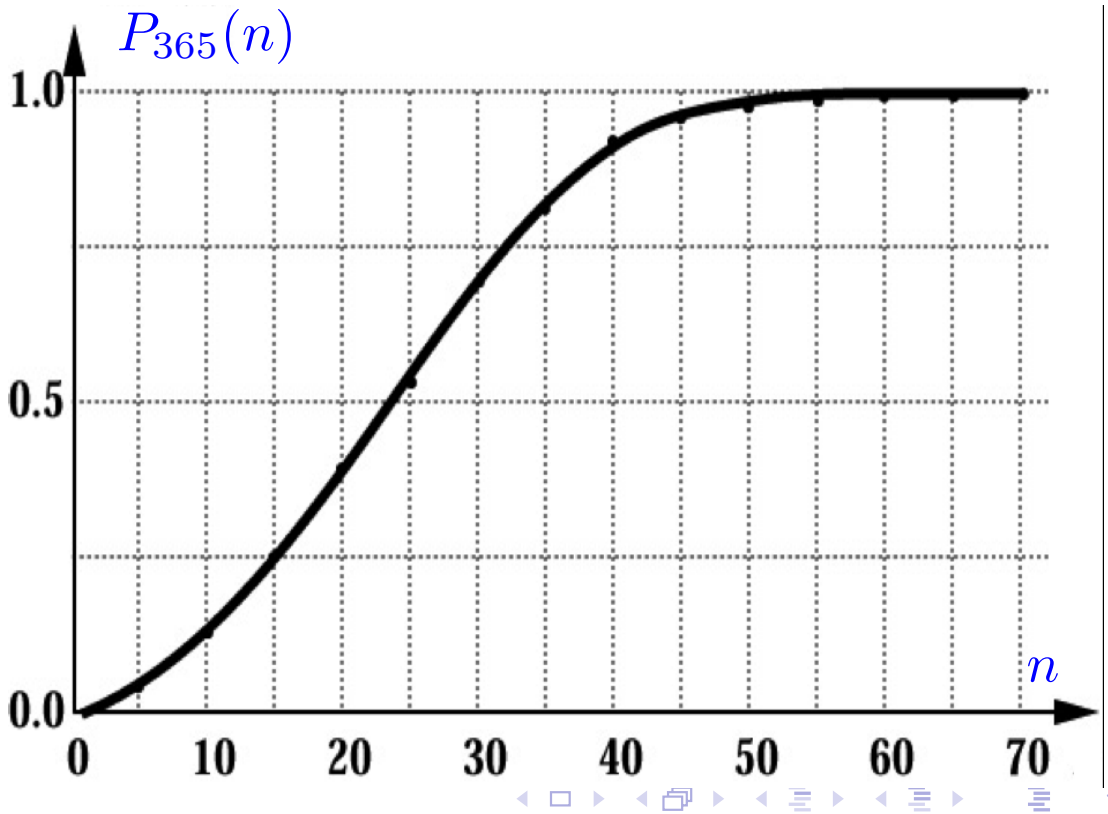
$$\boxed{Q(m,n) = \frac{m!}{m^n(m-n)!};} \qquad 1 \le n \le m$$

Probability of One or More Collisions

Probability, $P_m(n)$, of at least one collision between n items, being randomly tossed into the table with m slots:

$$P_m(n) = 1 - Q(m,n) = 1 - \frac{m!}{m^n(m-n)!}$$

n	%	$P_{365}(n)$
10	2.7	0.1169
20	5.5	0.4114
30	8.2	0.7063
40	11.0	0.8912
50	13.7	0.9704
60	16.4	0.9941



Open Addressing with Linear Probing (OALP)

The simplest collision resolution policy:

- Successive search for the first empty entry at a lower location
- If no such entry, then wrap around the table

Lemma 3.31:

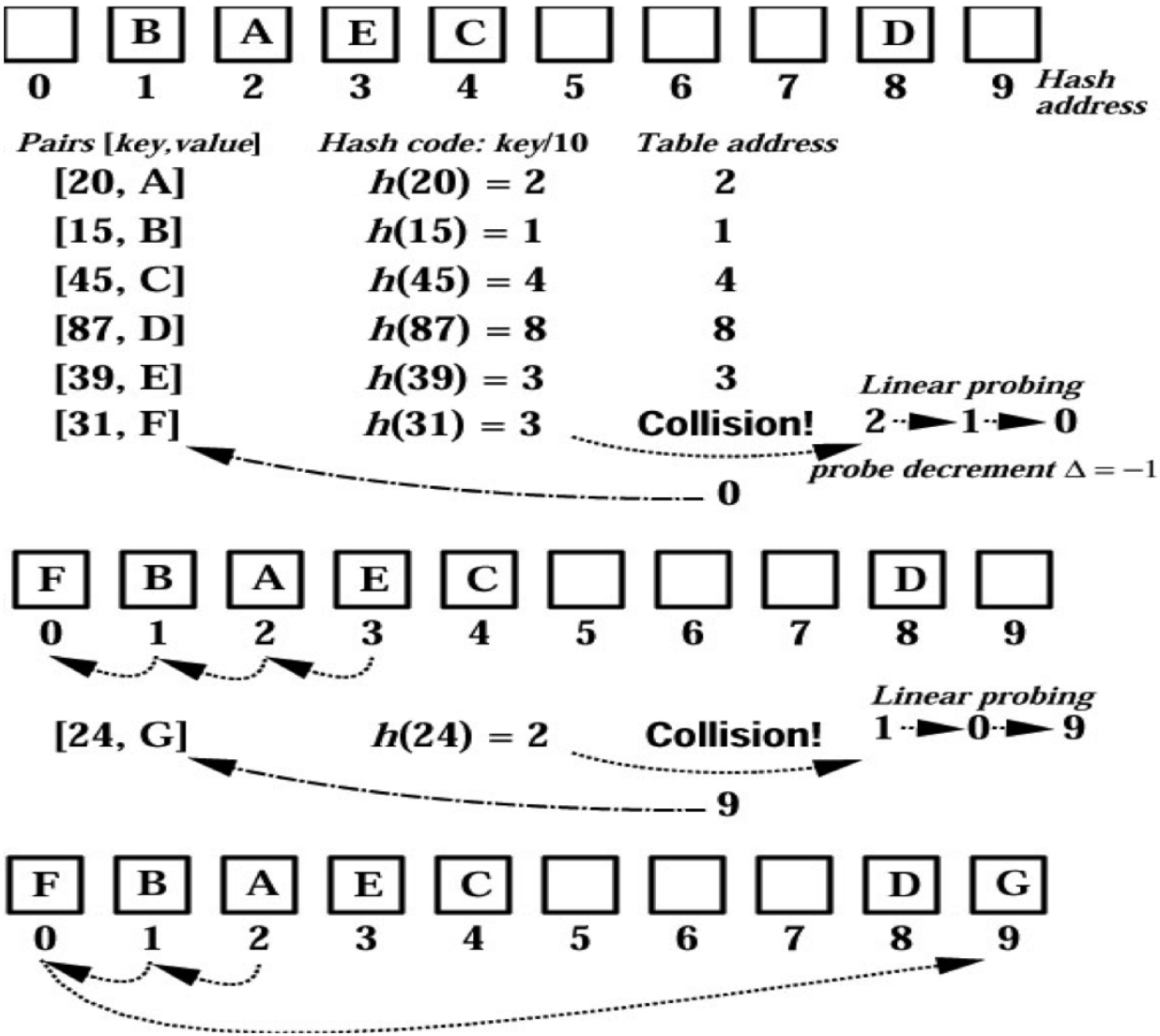
Assuming uniformly hashed random input, the expected number of probes for successful, $T_{\text{ss}}(\lambda)$, and unsuccessful, $T_{\text{us}}(\lambda)$, search in a hash table using OALP depends on load factor $\lambda = \frac{n}{m}$ as

$$T_{\text{ss}}(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \text{ and } T_{\text{us}}(\lambda) = \frac{1}{2}\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right),$$

respectively.

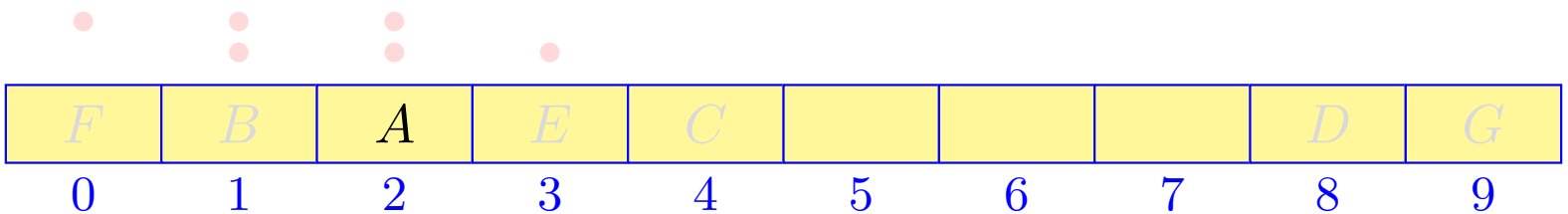
The proof is beyond the scope of this course.

OALP Example: $n = 5..7$; $m = 10$



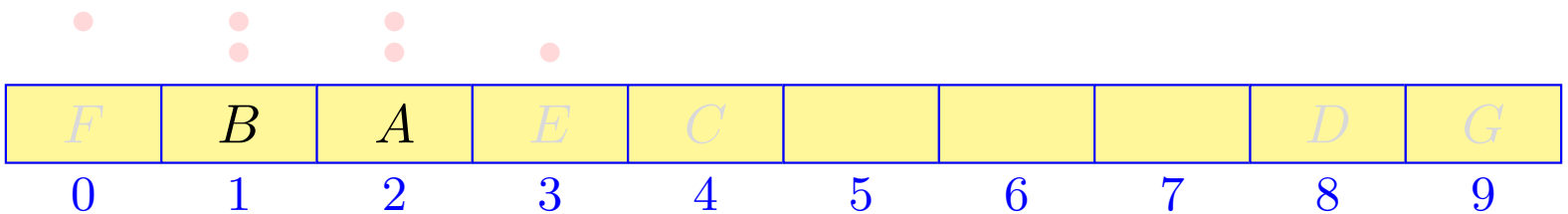
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



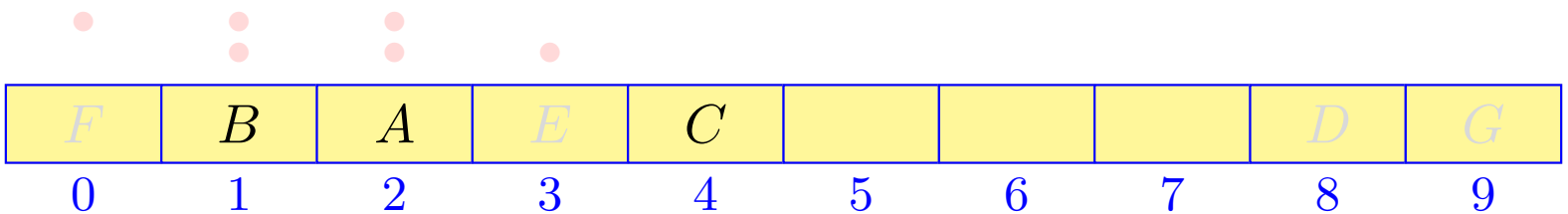
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



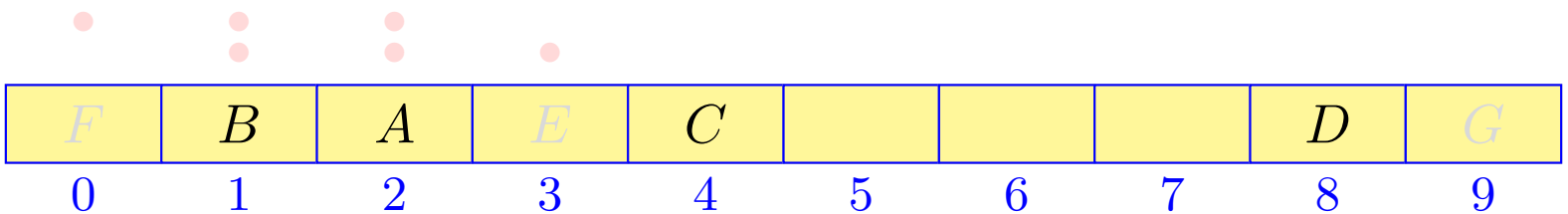
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



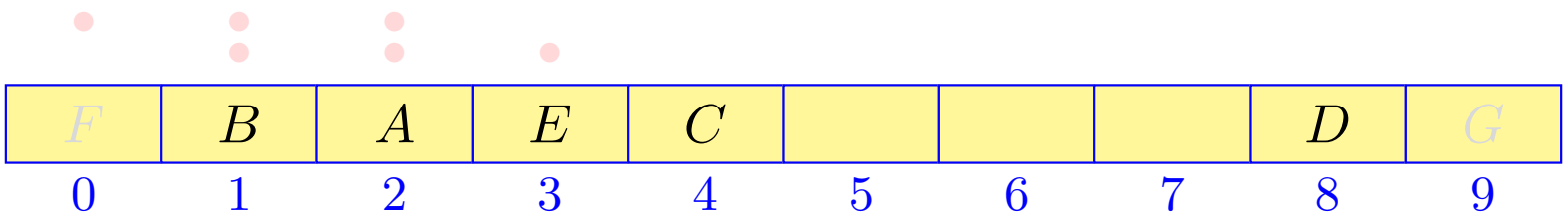
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



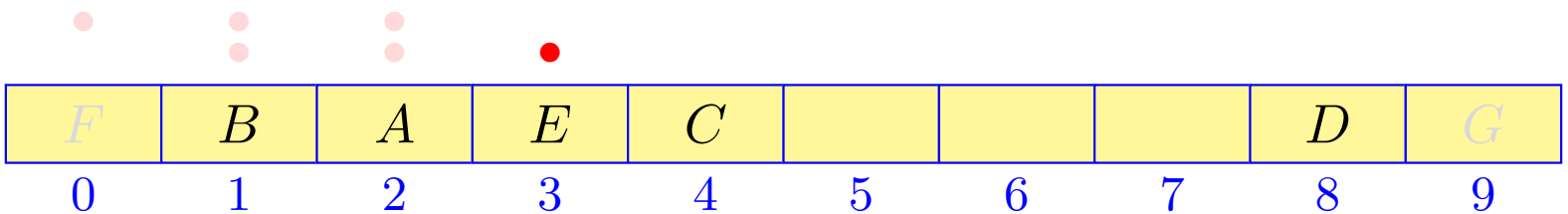
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



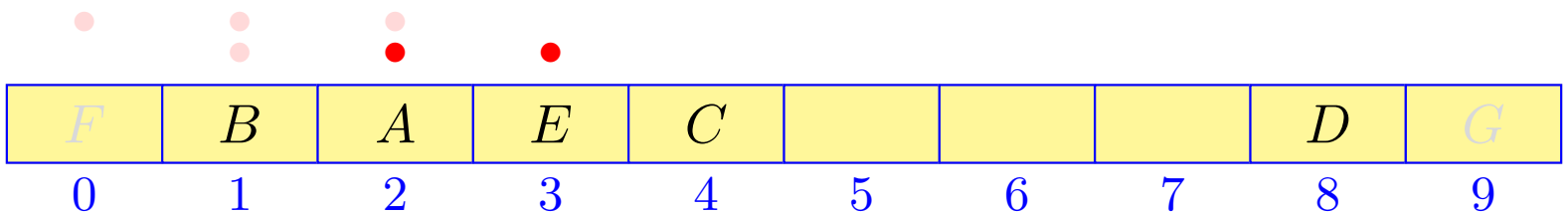
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



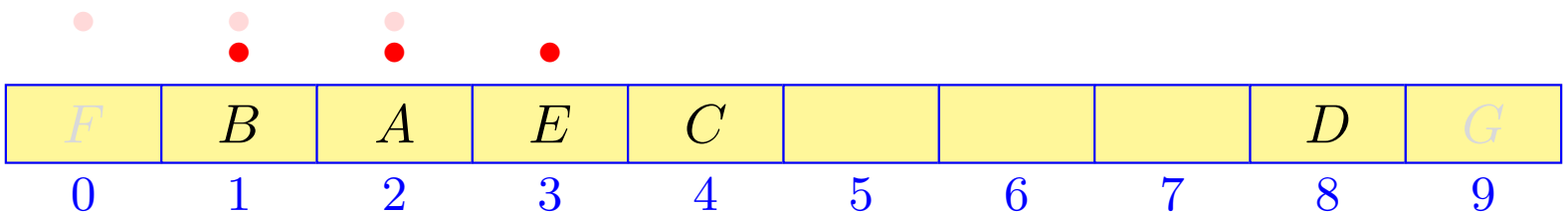
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



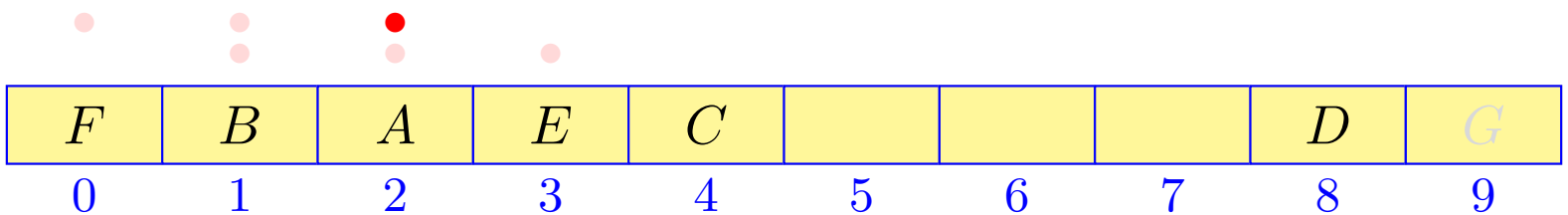
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7

F	B	A	E	C				D	G
0	1	2	3	4	5	6	7	8	9

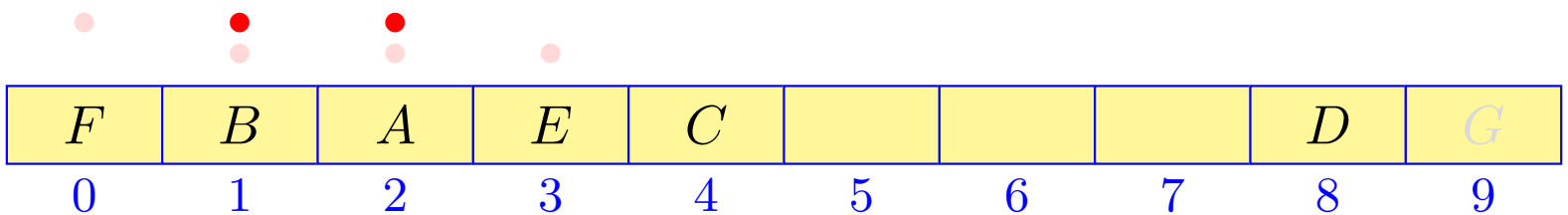
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



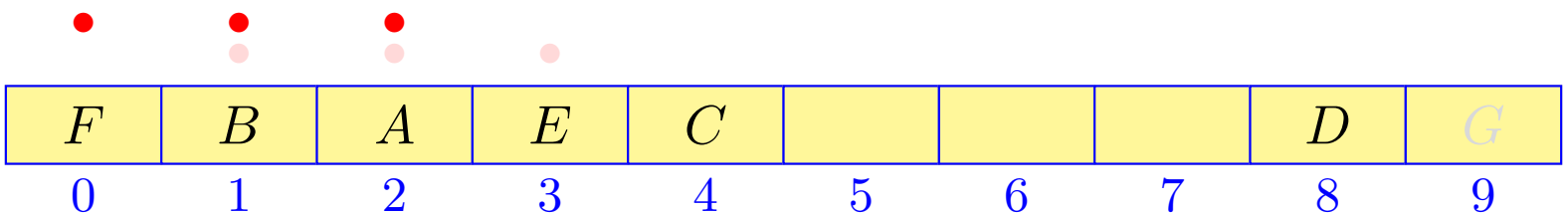
OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



OALP Example: $n = 5..7$; $m = 10$

Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7



OALP Example: $n = 5..7$; $m = 10$

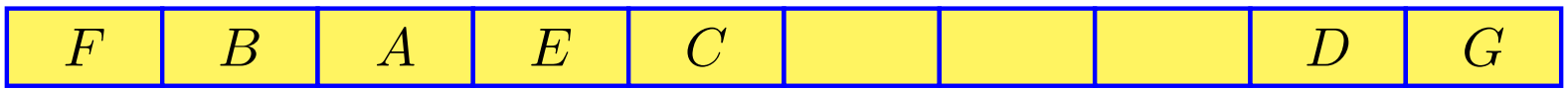
Data: $[k, v]$	**Hash**: $h(k) = \lfloor \frac{k}{10} \rfloor$	**Address**: a_k	**Probes**	# of probes	λ
$[20, A]$	2	2	2	1	0.1
$[15, B]$	1	1	1	1	0.2
$[45, C]$	4	4	4	1	0.3
$[87, D]$	8	8	8	1	0.4
$[39, E]$	3	3	3	1	0.5
$[31, F]$	**3**	0	3, 2, 1, 0	4	0.6
$[24, G]$	**2**	9	2, 1, 0, 9	4	0.7

F	B	A	E	C				D	G
0	1	2	3	4	5	6	7	8	9

OALP: Pros and Cons

Pro: Simple to implement.

Con: Degeneration of hash tables due to **clustering**.



- **Cluster** – a sequence of adjacent occupied table entries.
- Tendency to form clusters around locations with one or more collisions.
- The larger the cluster, the faster its growth.
- Longer search for an empty address due to clustering:
  - Average number ν of probes for Example in Slides 17–18:

λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7
ν	1	1	1	1	1	1.5	1.86

Another probing scheme, **double hashing**, reduces the likeihood of clustering.

Open Addressing with Double Hashing (OADH)

Better collision resolution policy reducing the clustering:

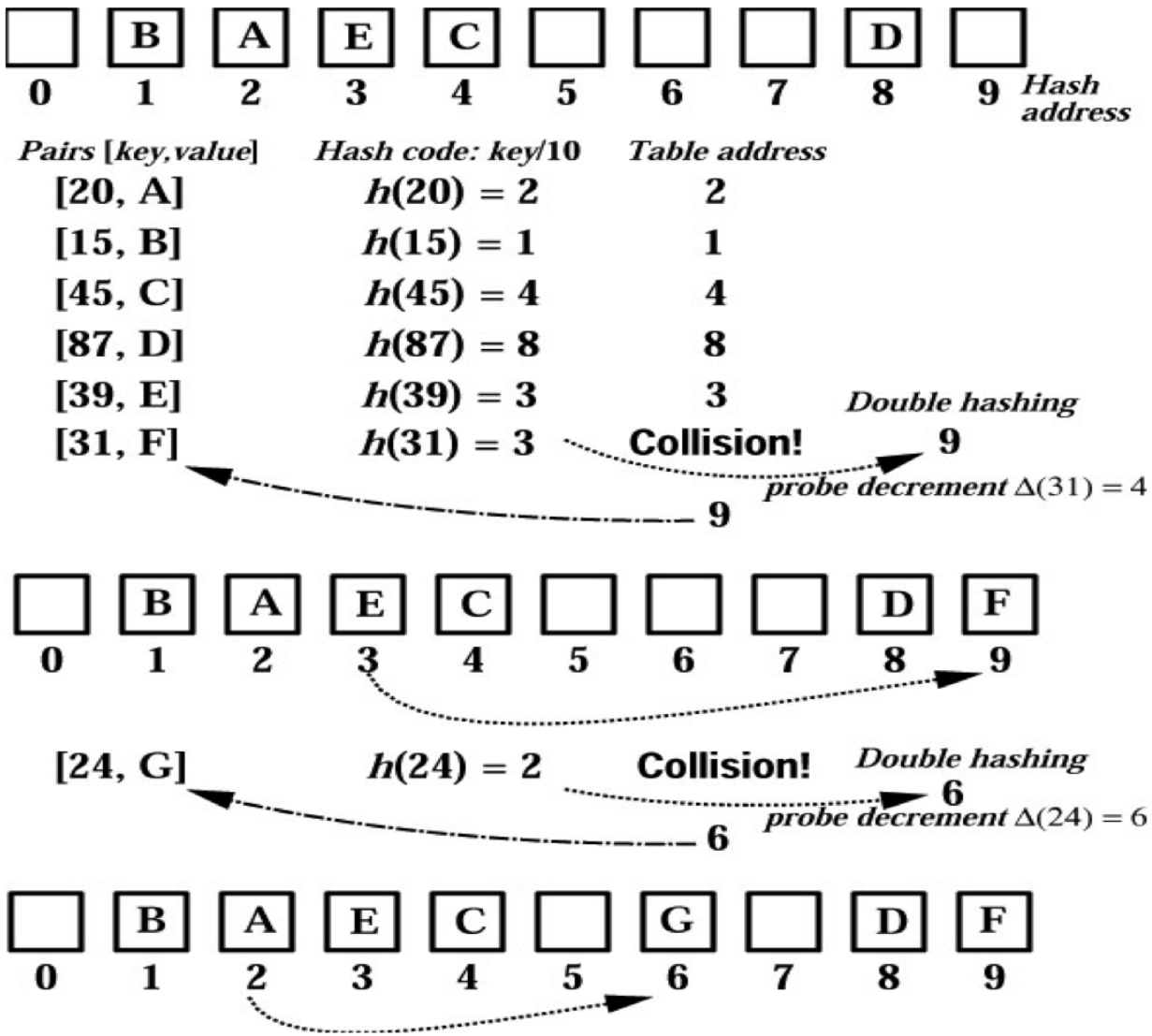
- hash the collided key again with a different hash function
- use the result of the second hashing as an increment for probing table locations (including wraparound)

Lemma 3.30:

Assuming that OADH provides nearly uniform hashing, the expected number of probes for successful, $T_{\text{ss}}(\lambda)$, and unsuccessful, $T_{\text{us}}(\lambda)$, search in a hash table with load factor $\lambda = \frac{n}{m}$ is, respectively:

$$T_{\text{ss}}(\lambda) = \frac{1}{\lambda} \ln\left(\frac{1}{1-\lambda}\right) \text{ and } T_{\text{us}}(\lambda) = \frac{1}{1-\lambda}$$

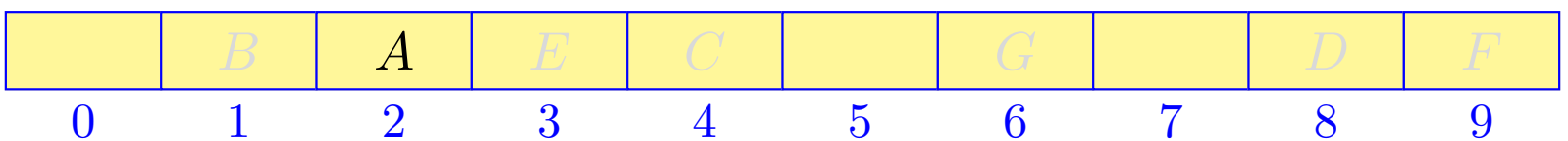
OADH Example: $n = 5..7$; $m = 10$



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

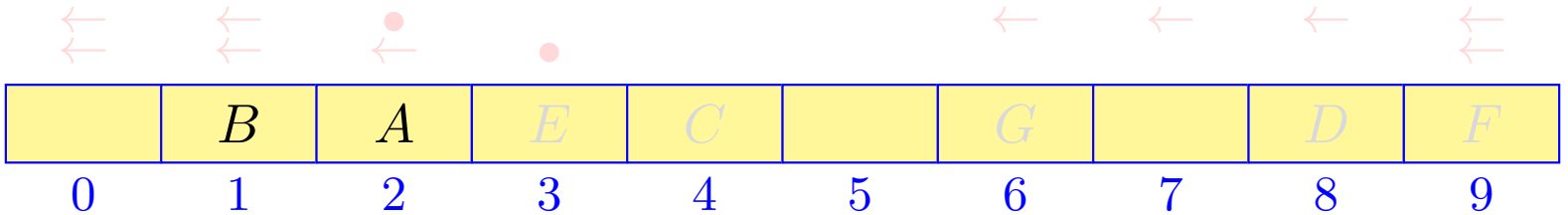
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

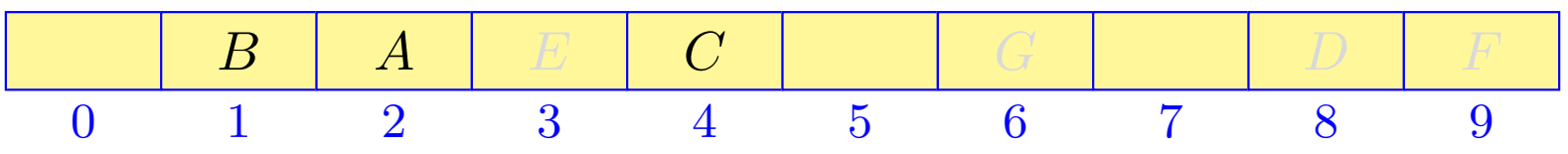
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

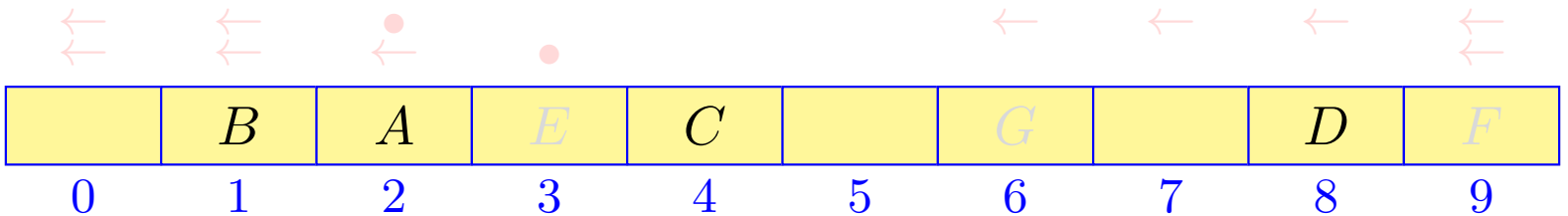
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

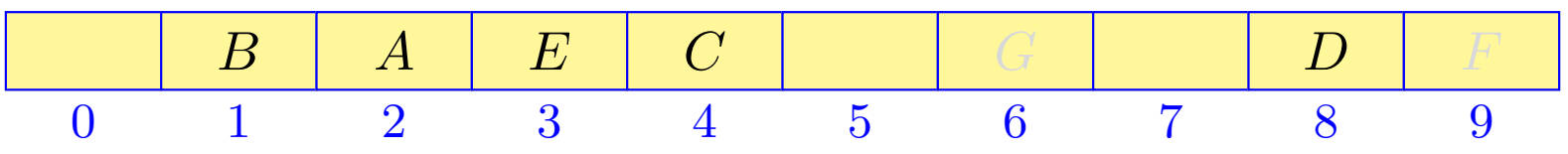
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

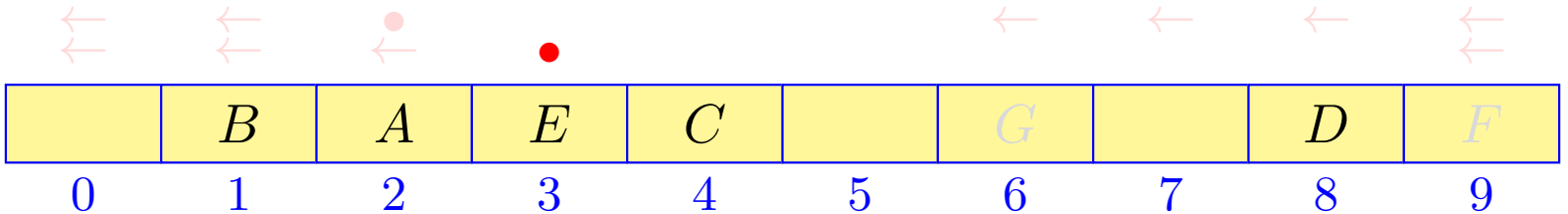
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

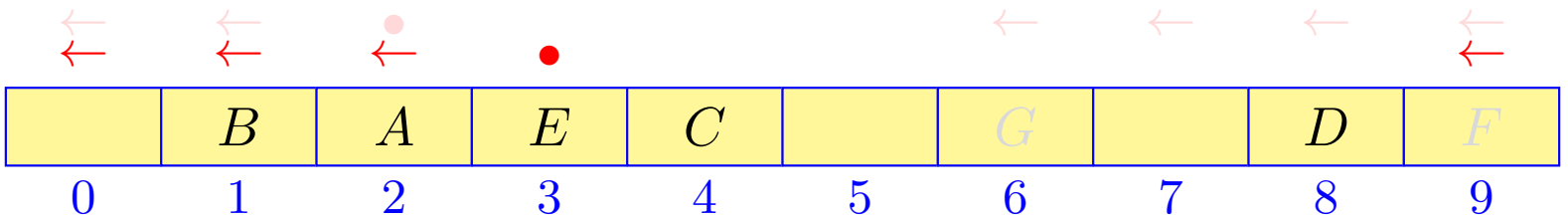
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

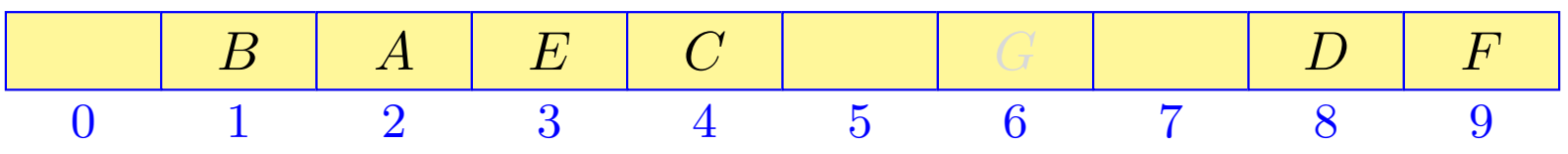
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

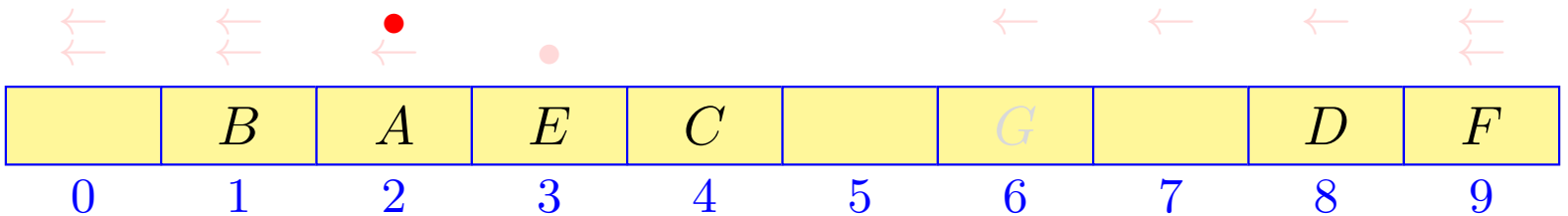
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

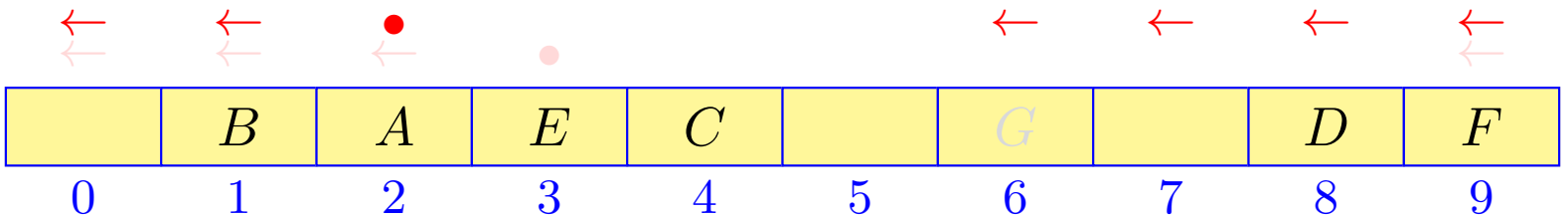
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

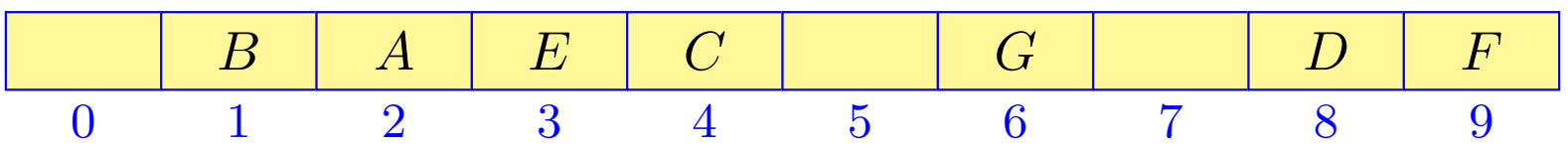
Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH Example: $n = 5..7$; $m = 10$

Hash: $h(k) = \lfloor \frac{k}{10} \rfloor$; **second hash**: $\Delta(k) = (h(k) + k) \mod 10$

Data: $[k, v]$	$h(k)$	**Address**:	$\Delta(k)$	**Probes**	# of probes	λ
$[20, A]$	2	2		2	1	0.1
$[15, B]$	1	1		1	1	0.2
$[45, C]$	4	4		4	1	0.3
$[87, D]$	8	8		8	1	0.4
$[39, E]$	3	3		3	1	0.5
$[31, F]$	**3**	9	$\Delta(31) = \mathbf{4}$	3, 9	2	0.6
$[24, G]$	**2**	6	$\Delta(24) = \mathbf{6}$	2, 6	2	0.7



OADH vs. OALP: More Uniform Hashing

More clusters than OALP, but of smaller sizes:

OADH

	B	A	E	C		G		D	F

OALP

F	B	A	E	C				D	G

- OALP: One custer "CEABFGD" of size 7.
- OADH: 3 clusters "CEAB", "G", and "FD" of sizes 4, 1, and 2, respectively.
- Unlike OALP, OADH does not extend clusters only at one end.
- Unlike OALP, OADH does not tend to join nearby clusters.
- Average number ν of probes in the above examples:

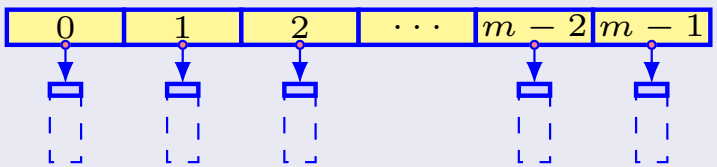
λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7
OALP : ν	1	1	1	1	1	1.5	1.86
OADH : ν	1	1	1	1	1	1.17	1.29

Two More Collision Resolution Techniques

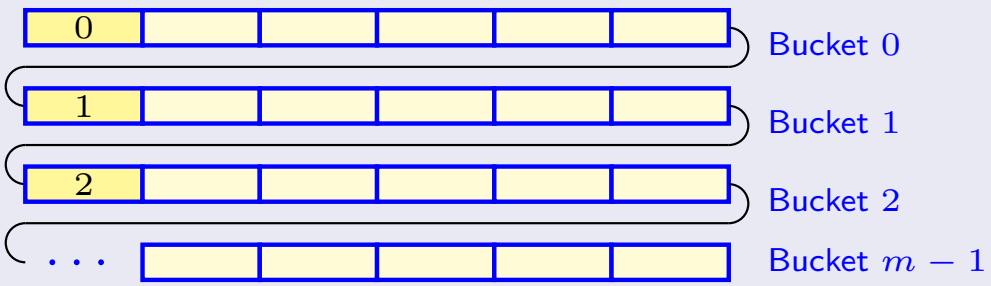
Main problem of open addressing: if significant number of items need to be deleted – all logically deleted items must remain in the table until the table can be re-organised.

Two techniques to attenuate this drawback:

- **Separate chaining** – a hash table with m addresses of linked lists

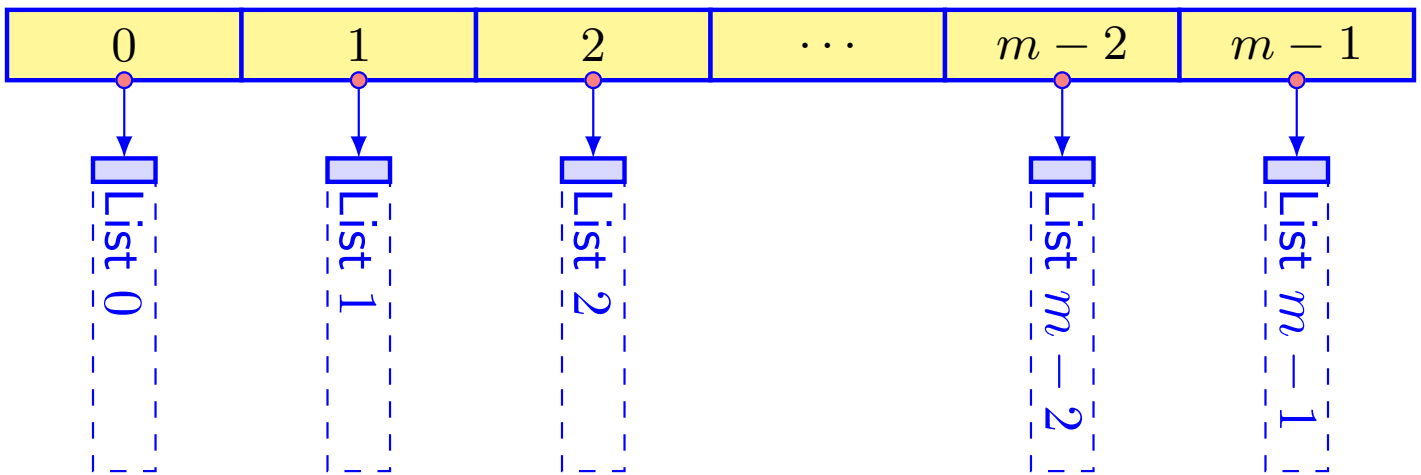


- **Hash bucket** – a large table of m buckets with hash addresses:



Separate Chaining

Hash table with m addresses $h(k) \in \{0, 1, \ldots, m-1\}$



All keys $k_1, k_2, \ldots,$ with have collided at a single hash address, $a = h(k_1) = h(k_2) = \ldots,$ are placed into a linked list, or *chain*, started at that address, a.

Separate Chaining: Expected Running Time (Lemma 3.29)

Assuming the hash address, $h(k)$, for a key k is computed by one elementary operation, the expected running time for unsuccessful search in a hash table with load factor λ using separate chaining is

$$T_{\text{us}}(\lambda) = 1 + \lambda$$

The expected running time for successful search is

$$T_{\text{us}}(\lambda) \in \mathrm{O}\left(1 + \frac{\lambda}{2}\right)$$

Update, retrieval, insertion and deletion all take time

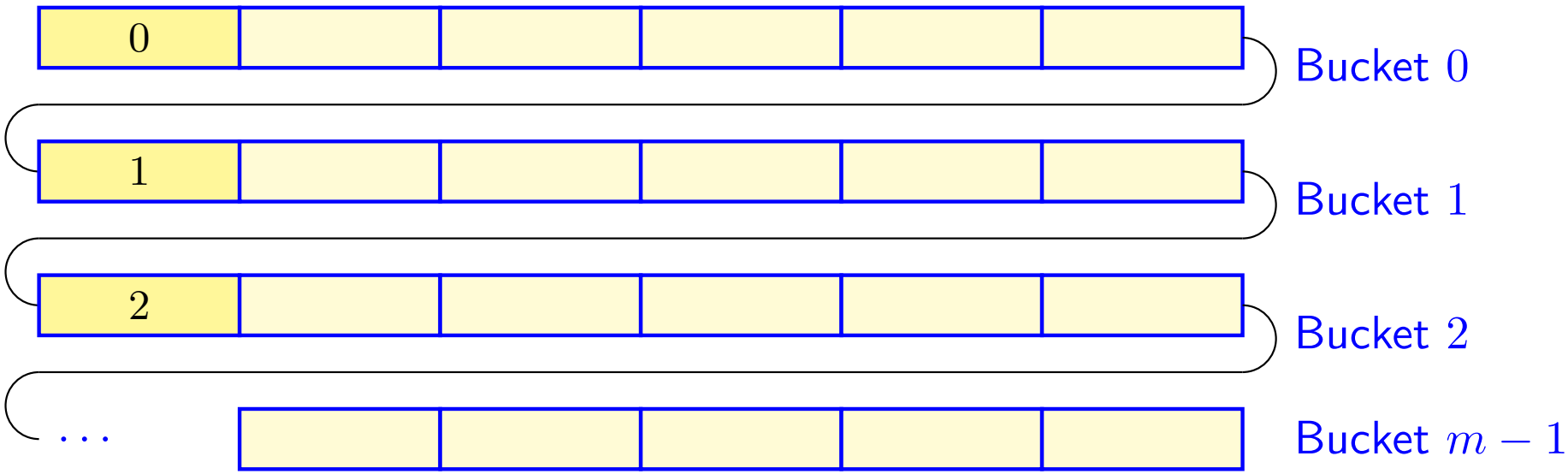
$$T(\lambda) \in \mathrm{O}(1 + \lambda)$$

Separate Chaining: Expected Running Time

Proof.

- The average running time to unsuccessfully search for the key at that address is equal to the average length of the associated chain, $\lambda = \frac{n}{m}$.
- Thus in total $T_{\text{us}}(\lambda) = 1 + \lambda = 1 + \frac{n}{m}$.
- The average value of $T_{\text{ss}}(\lambda)$ (in accord with Lemma 3.3) equals one plus the average of the times for the unsuccessful searches undertaken while building the table.
- In this case, it is one plus the average of $0 + 1 + 2 + \ldots + \lambda$, i.e., $1 + \frac{\lambda}{2}$.
- Since update, retrieval and deletion from a linked list take constant time, $T_{\text{ss}}(\lambda)$ is in $\mathrm{O}\left(1 + \frac{\lambda}{2}\right)$.

Hash Bucket



- A big hash table is divided into m small sub-tables, or *buckets*.
- The hush function, $h(k)$, maps a key k onto one of the buckets.
- Collided keys are stored in each bucket a sequentially in their ascending order: $k_1 < k_2 < \ldots$ if $a = h(k_1) = h(k_2) = \ldots$.

Implementation of Hashing: Resizing

Resizing a hash table for open addressing to keep $\lambda < 0.75$:

- Doubling the table size when $\lambda \geq 0.75$.
- Readdressing all the elements with a new hash function.
- Average insertion time: $\Theta(1)$ (as now $\lambda \ll 1$).
- Time for m insertions – of order m.
- Total time to grow a table from 1 to $m = 2^k$ elements: $1 + 2 + \ldots + 2^{k-1} = 2^k - 1 = m - 1$.

Thus, the average insertion time with resizing is still $\Theta(1)$.

Implementation of Hashing: Deletion

Deletion of a table entry: simple for separate chaining (SC), but difficult for open addressing (OA) tables.

Because an emptied place in an OA table invalidates the search for subsequent keys, make the table valid as follows:

- Mark the deleted entries empty for insertion, but non-empty for key search.
- If too many such marked entries, really delete them and resize the table.

The deletion time $\mathrm{O}(1)$ for both SC and OA hash tables.

Choosing a Hash Function, $h(k)$

Ideal hash function: uniform and random mapping onto all table addresses.

- Similar to generating uniformly distributed pseudorandom numbers.

Perfect hash function for a fixed set of keys:

- One-to-one mapping of the set of keys onto a set of table indices.
- Main design problem: quick computation with no large tables.
- Limited practical interest as data sets are not static and sets of keys cannot be predefined.

Four basic methods for non-fixed sets of keys:

- Division, folding, middle-squaring, and truncation

Choosing a Hash Function: Division

- Choose a prime number as the table size m.
  - Non-prime m: some slots may be unreachable for probing with double hashing.
- Convert keys, k, into integers.
- Use the remainder $h(k) = k \bmod m$ as a hash value of k.
- Get the double hashing decrement using the quotient $\left\lfloor \frac{k}{m} \right\rfloor$:

$$\Delta(k) = \max\left\{1, \left\lfloor \frac{k}{m} \right\rfloor \bmod m\right\}$$

Example of division: $k = 013402122$; $m = 10007$

$$\begin{aligned} h(k) &= 013402122 \bmod 10007 &&= 2749 \\ \Delta(k) &= \max\left\{1, \left\lfloor \frac{013402122}{10007} \right\rfloor \bmod 10007\right\} &&= 1339 \end{aligned}$$

Choosing a Hash Function: Folding

- Divide the integer key, k, into sections.
- Add, subtract, and / or multiply them together for combining into the final value, $h(k)$

Examples of folding: $k = 013402122$

$$k \to 013, 402, 122 \quad \to h(k) = 013 + 402 + 122 = 537$$

$$k \to 0134, 0212, 2 \quad \to h(k) = -0134 + 0212 \times 2 = 290$$

$$k \to 01, 34, 02, 12, 2 \to h(k) = 01 \times 34 + 02 \times 12 \times 2 = 082$$

Choosing a Hash Function: Middle-squaring

- Choose a middle section of the integer key, k.
- Square the chosen section.
- Use a middle section of the result as $h(k)$

Examples of middle-squaring: $k = 013402122$

$k \rightarrow \text{mid: } 402 \quad \rightarrow 402^2 = 161404 \quad \rightarrow \text{mid: } h(k) = 6140$

$k \rightarrow \text{mid: } 34021 \rightarrow 34021^2 = 1157428441 \rightarrow \text{mid: } h(k) = 7428$

Choosing a Hash Function: Truncation

- Delete part of the key, k.
- Use the remaining digits (bits, characters) as the hash address, $h(k)$.

Examples of truncation:

$$k = 013402122 \quad \rightarrow \quad \text{last 3 digits: } h(k) = 122$$

$$k = 013402122 \quad \rightarrow \quad 2^{\text{nd}}, 4^{\text{th}}, \text{ and last digit: } h(k) = 142$$

Notice that truncation does not spread keys uniformly into the table; thus it is often used in conjunction with other methods.

Universal Class of Hash Functions

Like with `quicksort`, randomisation can protect hashing against bad worst-case behaviour of any fixed hash function.

Definition 3.33: Universal class of hash functions

Notation:

- K – a set of keys.
- m – a size of a hash table with indices $M = \{0, 1, \ldots, m-1\}$.
- F – a set of the hash functions h, mapping K to M.

F is a **universal class** if any distinct pair of keys, $k, \kappa \in K$ collide for no more than $\frac{1}{m}$ of the functions in the class F:

$$\frac{1}{|F|} |\{h \in F | h(k) = h(\kappa)\}| \leq \frac{1}{m}$$

Any pair of keys collide with a probability of at most $\frac{1}{m}$ under the random selection of a function from the class.

Popular Universal Class of Hash Functions

Conditions:

- A set $K = \{0, 1, \ldots, p-1$ of integer keys, such that its cardinality $p = |K|$ is a prime number.
- An arbitrary hash table size m.

.

Theorem 3.34 (Universal Class of Hash Functions)

Let $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$ for any pair of integers $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$.

Then $F = \{h_{a,b} \,|\, 1 \leq a < p \text{ and } 0 \leq b < p\}$ is a universal class.

It is proven (see Textbook) that for any distinct pair of keys, (k, κ), at most $\frac{1}{m}$-th fraction of the hash functions in the set F cause these keys to collide.

Strategy for Choosing a Hash Function

Theorem 3.34 suggests how to choose a hash function at run time:

1. Find the current size of the set of keys to hash.
2. Select the next prime number, p, larger than the size of the key set found.
3. Randomly choose integers a and b such that $0 < a < p$ and $0 \leq b < p$.
4. Use the function $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$.

Example: 990 keys; $p = 991$; $m = 701$;

$h_{5,800}(k) = ((5k + 800) \bmod 991) \bmod 701 \Rightarrow$

Key	10	700	55	330
Index	149	336	84	468

e.g.,

$$h_{5,800}(330) = ((\underbrace{5 \times 330}_{1650} + 800) \bmod 991) \bmod 701 = ((\underbrace{2450 \bmod 991}_{468}) \bmod 701 = 468$$

Efficiency of Search in Hash Tables

Load factor λ:
if a table of size m has exactly n occupied entries, then $\lambda = \frac{n}{m}$.

Average numbers of probe addresses examined for a successful, $T_{\text{ss}}(\lambda)$, and unsuccessful $T_{\text{us}}(\lambda)$, search depend on λ::

	OALP : $0 \leq \lambda < 0.7$	OADH : $0 \leq \lambda < 0.7$	SC : $\lambda = 4^*$
$T_{\text{ss}}(\lambda)$	$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$	$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$	$1 + \frac{\lambda}{2}$
	≤ 2.15	≤ 1.70	3
$T_{\text{us}}(\lambda)$	$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$	$\frac{1}{1-\lambda}$	$1 + \lambda$
	≤ 6.05	≤ 3.33	5

$*$) SC – separate chaining: λ may be greater than 1.

Average Search Time in Hash Tables

λ	Successful search $T_{\text{ss}}(\lambda)$			Unsuccessful search $T_{\text{us}}(\lambda)$		
	OALP	OADH	SC	OALP	OADH	SC
0.10	1.06	1.05	1.05	1.12	1.11	1.10
0.25	1.17	1.15	1.13	1.39	1.33	1.25
0.5	1.5	1.39	1.25	2.5	2.0	1.5
0.7	2.15	1.7	1.35	6.05	3.33	1.7
0.75	2.5	1.85	1.38	8.5	4.0	1.75
0.90	5.5	2.56	1.45	50.5	10.0	1.90
0.99	50.5	4.65	1.49	**5000.0**	100.0	1.99

Table Representations: Comparative Performance

Operation	Representation of a table of size m		
	Sorted array	AVL tree	Hash table
Initialise	$O(m)$	$O(1)$	$O(m)$
Is full?	$O(1)$	$O(1)$	$O(1)$
Search*	$O(\log m)$	$O(\log m)$	$O(1)$
Insert	$O(m)$	$O(\log m)$	$O(1)$
Delete	$O(m)$	$O(\log m)$	$O(1)$
Enumerate	$O(m)$	$O(m)$	$O(m \log m)^{\circ}$

$*$) also: Retrieve, Update

$\circ$) Entries of a hash table must be sorted first in ascending order (it takes the linearithmic time).