Symbol Tables

Hash Tables

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Definition

- Hash-tables
  - An ADT used to insert, search, delete, **not** to order or to select keys
  - Reduce the storage requirements of direct-access tables from $\theta(|U|)$ to $\theta(|K|)$
- Efficiency
  - Memory usage in the order of the number of keys stored in that table (not in the order of $|U|$)
    - $M(K) = \theta(|K|)$
  - Average access is constant time
    - $T(K) = O(1)$

$|K|$ = Forecast number of keys to be stored
$|U|$ = Number of keys in the key universe
Usually $|K| \ll |U|$

Definition

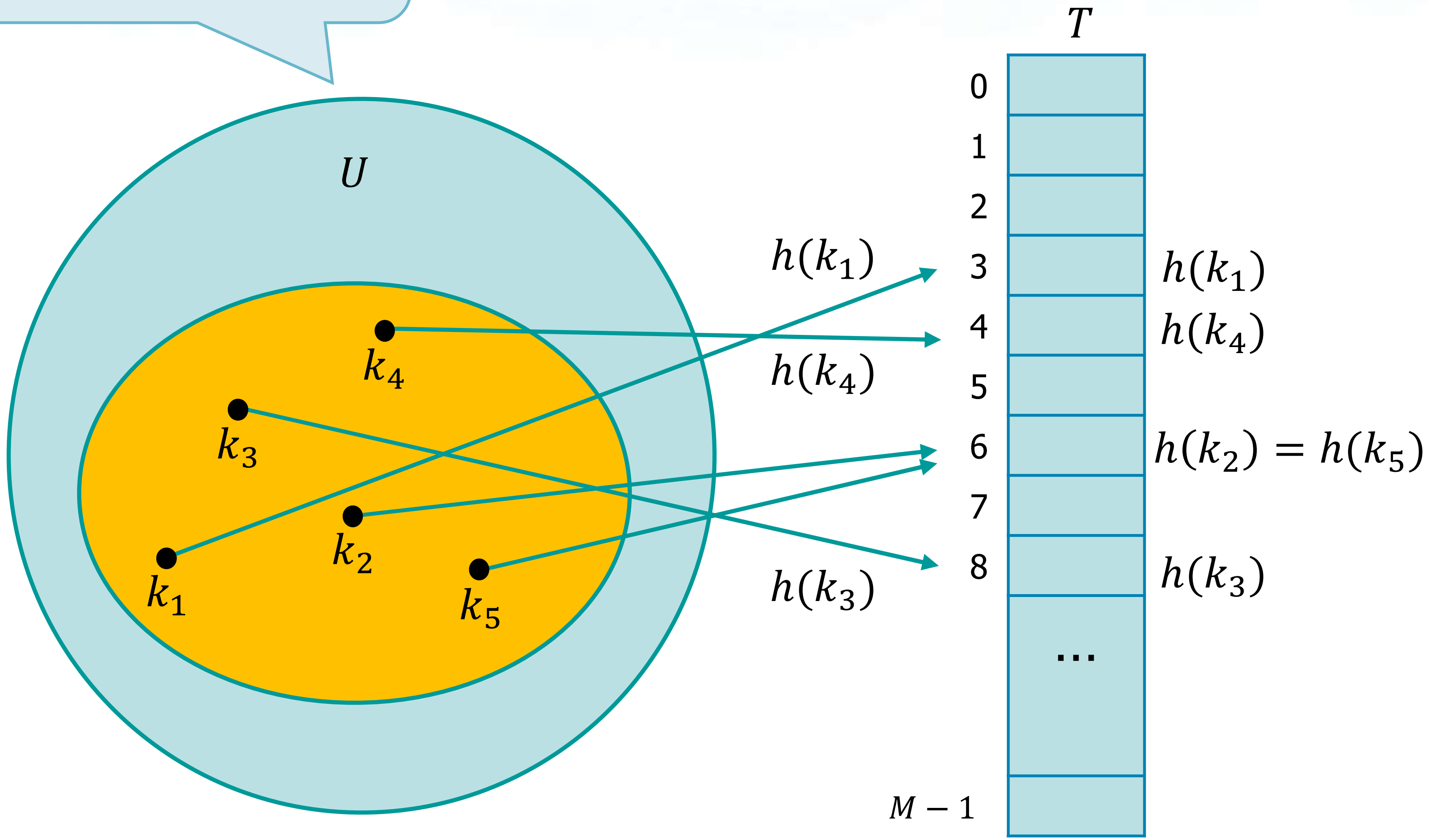
- It uses
  - A table (an array) to store the data
  - A function to transform each key into its position (index) into an array
- The table
  - Has size M and stores $|K|$ elements
    - $|K| \ll |U|$
  - Has addresses (indices) in the range $[0, M-1]$

Previously **st**

Previously **getindex**

Hash Function

The mapping between $k \in U$ and elements in the table is $|U|: M$ (not $1:1$)



Hash Function

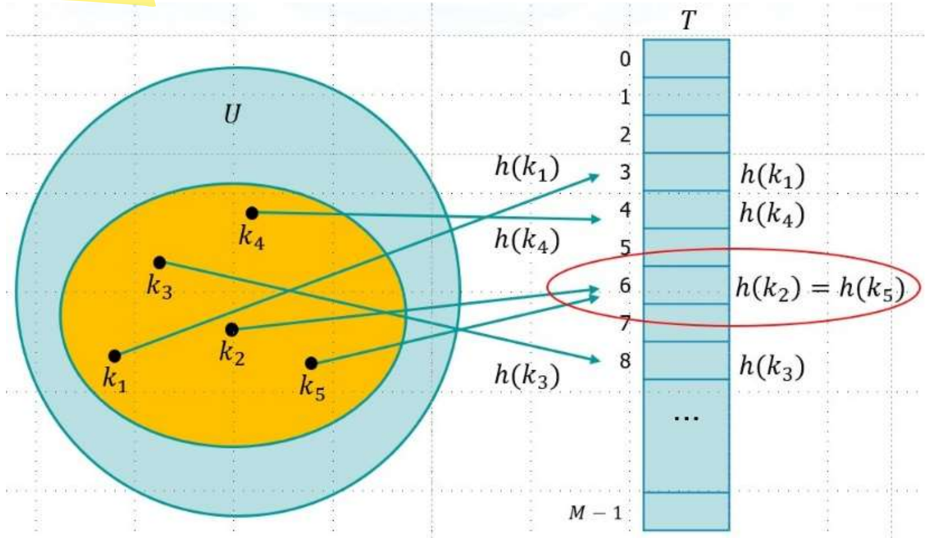
- The function used to map a key into an array index (position) is called **hash function**
  - It transforms the search key into a table index, i.e., it creates a correspondence between a key k and a table address $\mathrm{h}(k)$

$$h\colon U \rightarrow \{0, 1, 2, \cdots, M-1\}$$

  - Each element of key k is stored at the address $\mathrm{h}(k)$
  - As $|K| \ll |U|$ the hash function creates a mapping which is $\mathrm{n}\colon 1$, no more $1\colon 1$ as in the direct access tables

Hash Function

- Every time two different keys are placed in the same table element we have a conflict
  - Such a conflict is called a **collision**
- Collisions may always happen as the
  - Hash tables map $|U|$ elements into $|M|$ slots
    - The table cannot contain all keys within the U
  - No hash function is perfect
    - The mapping may always create conflicts

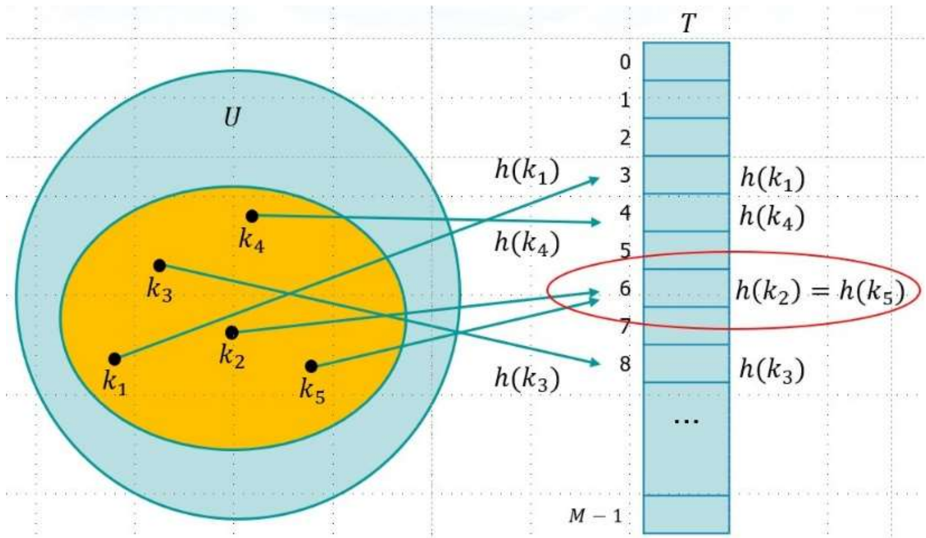


Hash Function

- Collisions in hash tables imply
  - **Designing** proper hash functions to **minimize** collisions we must
  - **Dealing** with the **remaining** collisions

Problem # 1

Problem # 2



Problem 1: Designing a hash function

- If the k keys are equiprobable, then the $h(k)$ values must be equiprobable
  - Practically, the k keys are not equiprobable, as they are correlated
- To make the $h(k)$ values equiprobable it is necessary to
  - Distribute $h(k)$ in a uniform way
  - Make $h(k_i)$ uncorrelated from $h(k_j)$
  - Uncorrelate $h(k)$ from k
  - "Amplify" differences
- Hash function can be designed in different ways

For example, Italian first names

The Multiplication Method

- If keys k are floating point numbers

Key's range

$$k \in [s, t]$$
$$h(k) = \left\lfloor \frac{(k-s)}{(t-s)} \cdot M \right\rfloor$$

$\lfloor \quad \rfloor$ = floor = largest integer smaller than

- Example

$$M = 97$$
$$k \in [0, 1.0] = 0.513871$$
$$h(k) = \left\lfloor \frac{(k-s)}{(t-s)} \cdot M \right\rfloor = \left\lfloor \frac{(0.513871 - 0)}{(1.0 - 0)} \cdot 97 \right\rfloor = 49$$

(note that in reality, the numbers used are on a much greater scale)

The Multiplication Method

- Implementation

```
int hash (float k, int M) {
  return ( ( (k-s)/(t-s) ) * M);
}
```

The Module Method

- If keys k are integer numbers

Fast and easy to compute

$$k \in integers$$
$$h(k) = k \ \% \ M$$

Alternative method

or

$$k \in integers$$
$$h(k) = 1 + k \ \% \ \widehat{M} \quad with \quad \widehat{M} < M$$

- Examples

$$M = 19$$
$$k = 11 \quad \rightarrow \quad h(k) = 11 \ \% \ 19 = 11$$
$$k = 31 \quad \rightarrow \quad h(k) = 31 \ \% \ 19 = 12$$
$$k = 29 \quad \rightarrow \quad h(k) = 29 \ \% \ 19 = 10$$

note that k=30 ->h(k)=30%19=11 --> collision

The Module Method

- Implementation

```
int hash (int k, int M) {
  return (k%M);
}
```

- It is convenient to use prime numbers for M to consider all digits/bits
  - If
    - $M = 2^n$ we use only the last n bits
    - $M = 10^n$ we use only the last n decimal digits
  - Keys will not evenly distribute

$k\%2^n$ gets the n LSBs of k
$k\%10^n$ gets the n LSDs of k

The Multiplication-Module Method

- If keys k are integer numbers

$$k \in integers$$
$$A \in]0,1[$$
$$h(k) = \lfloor k \cdot A \rfloor \,\%\, M$$

A is a constant value

  - A good value for A is

$$A = \frac{(\sqrt{5}-1)}{2} = 0.6180339887$$

The Multiplication-Module Method

- Examples

$$M = 19$$

$$A = \frac{(\sqrt{5}-1)}{2} = 0.6180339887$$

$$k = 11 \quad \rightarrow \quad h(k) = \lfloor 11 \cdot A \rfloor \,\%\, 19 = 6 \,\%\, 19 = 6$$

$$k = 31 \quad \rightarrow \quad h(k) = \lfloor 31 \cdot A \rfloor \,\%\, 19 = 19 \,\%\, 19 = 0$$

- Implementation

```
int hash (int k, int M) {
  return (((int) (k*A))%M);
}
```

Hash functions for short strings

- If keys k are **short** alphanumeric strings
  - The best strategy is to convert them into integers
  - Each string can be "evaluated" through a polinomial which "evalutes" the string as a number in a given base

$$N_{10} = 1234_{10} = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$
$$\text{Base b} = 10, \text{digits} = [0,9]$$

$$N_2 = 101101_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$\text{Base b} = 2, \text{digits} = [0,1]$$

  - Once the integer is obtained one of the previous strategies (i.e., the module method) can be applied

Hash functions for short strings

- Example

$$k = \text{now}$$
$$M = 19$$
$$h(k) = (p_{n-1} \cdot b^{n-1} + p_{n-2} \cdot b^{n-2} + \cdots + p_1 \cdot b^1 + p_0 \cdot b^0) \% M$$

$$h(\text{"}now\text{"}) = (p_2 \cdot b^2 + p_1 \cdot b^1 + p_0 \cdot b^0) \% 19 =$$
$$= (n \cdot 128^2 + o \cdot 128^1 + w \cdot 128^0) \% 19 =$$
$$= (110 \cdot 128^2 + 111 \cdot 128^1 + 119 \cdot 128^0) \% 19 =$$
$$= 1816567 \% 19 = 15$$

Polinomial interpretation of the string as a number in base b = 128

For each character we may use the corresponding ASCII value

Hash functions for long strings

- If keys are **long** alphanumeric strings
  - The previous computation overflows
    - Intermediate computations and result cannot be represented on a reasonable number of bits
  - It is possible to use the **Horner's** method
    - We rule-out M multiples after each step, instead of doing that at the end

$$h(k) = (p_{n-1} \cdot b^{n-1} + p_{n-2} \cdot b^{n-2} + \cdots + p_1 \cdot b^1 + p_0 \cdot b^0) \ \% \ M$$

$$h(k) = (\cdots(p_{n-1} \cdot b + p_{n-2}) \cdot b + p_{n-3}) \cdot b + \cdots + p_1) \cdot b + p_0) \ \% \ M$$

$$h(k) = (\cdots(p_{n-1} \ \% \ M) \cdot b + p_{n-2}) \ \% \ M) \cdot b + p_{n-3}) \ \% \ M) \cdot b + \cdots$$
$$\cdots + p_1) \ \% \ M) \cdot b + p_0) \ \% \ M$$

Hash functions for long strings

- Example

$$k = "averylongkey"$$
$$b = 128$$
$$h(k) = (p_{n-1} \cdot b^{n-1} + p_{n-2} \cdot b^{n-2} + \cdots + p_1 \cdot b^1 + p_0 \cdot b^0) \ \% \ M$$

$$h(k) = (97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^9 + 114 \cdot 128^8 + \cdots) \ \% \ M$$

$$h(k) = (\ldots(97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + \cdots) \ \% \ M$$

$$h(k) = (\ldots(97 \ \% \ M) \cdot 128 + 118) \ \% \ M) \cdot 128 + 101) \ \% \ M) \cdot 128) + \\ + 114) \ \% \ M) \cdot 128 + \cdots) \ \% \ M$$

Apply Horner's method

Hash functions for long strings

- Implementation

```
int hash (char *v, int M) {
  int h = 0;
  int base = 128;

  while (*v != '\0') {
    h = (h * base + *v) % M;
    v++;
  }

  return h;
}
```

Polinomial interpretation of the string as a number base $base = b = 128$

Hash functions for long strings

- To obtain a uniform distribution we must have a collision probability for two different keys equal to $^1/_M$
  - Base $b = 128 = 2^7$ is not a good base
- Rule of thumb to select b
  - A prime number
    - For example $b = 127$

```
int hash (char *v, int M) {
  int h = 0;
  int base = 127;
  while (*v != '\0') {
    h = (h * base + *v) % M;
    v++;
  }
  return h;
}
```

Hash functions for long strings

- Or even better random numbers different for each digit of the key
  - This approach is called **universal hashing**

```
int hash (char *v, int M) {
  int h = 0;
  int a = 31415, b = 27183;

  while (*v != '\0') {
    h = (h * a + *v) % M;
    a = ((a*b) % (M-1));
    v++;
  }

  return h;
}
```

Problem 2: Dealing with collisions

- A collision happens when

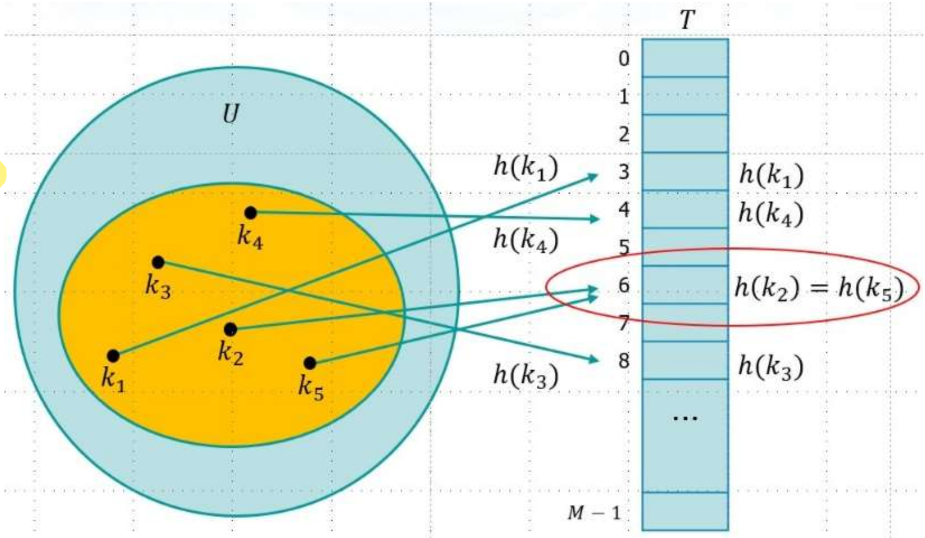
$$k_i \neq k_j \quad \rightarrow \quad h(k_i) = h(k_j)$$

- When a collision occur, we can deal with it adopting
  - Linear chaining
    - For each hash table entry, a list of elements stores all data items having the same hash function value

basic

  - Open addressing
    - For each collision, it places the same element somewhere else, i.e., in another table entry within the same table

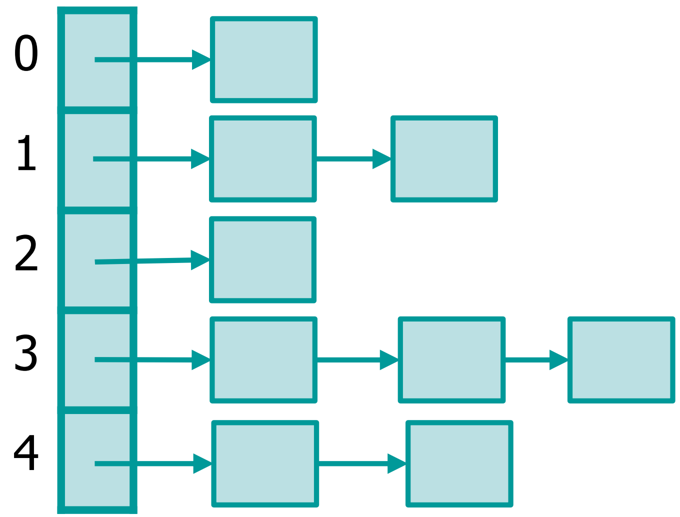
more complex



Method 1: Linear Chaining

Colllision --> put that element in a list

- More elements are stored in the same table location
  - An element does not contain a key anymore, but is points to a linked list including all elements which has the same hash function value
  - Each operation (insert/search/delete) must take the list into consideration

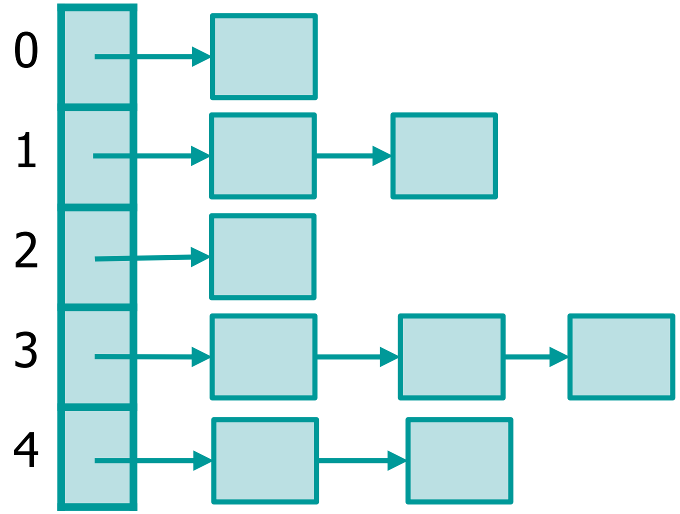


Method 1: Linear Chaining

- Insert
  - We must insert an element in the list
  - The most efficient approch is to insert new elements onto the list head
- Search
  - To search an element we must apply the hash function first and a list search after

but lists need to be very short (like 5 elements or so)

- Delete
  - To delete an element we must search it
    - Lists are not usually sorted as insertions are on the head
    - Delete it from the list



Method 1: Linear Chaining

- With linear chaining the hash table
  - Can be smaller than the number of elements $|K|$ that have to be stored in it
  - The smaller the table the longer the linked lists
    - Too long lists imply inefficiency
    - It is a good rule of thumb to have lists with an average length varying from 5 to 10 elements
    - Select M as the smallest prime larger than the maximum number of keys divided by 5 (or 10) such that the average list length would be 5 (or 10)

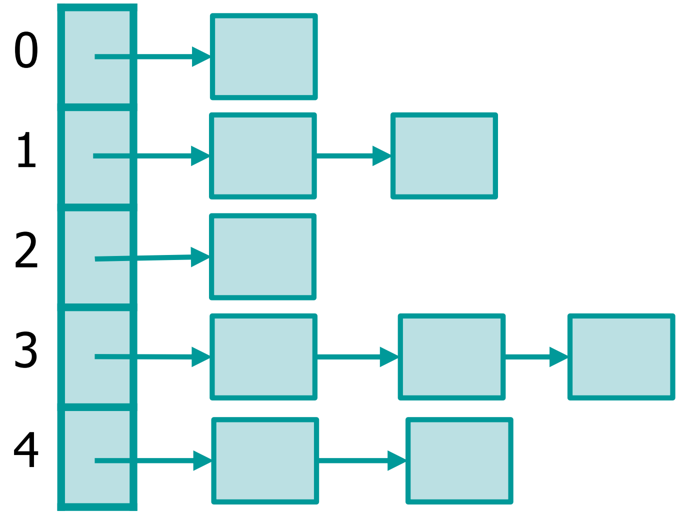
We must know (guess) the number of keys we want to store in the hash table before allocating it !

Method 1: Linear Chaining

- Given
  - N = number of stored elements
  - M = size of the hash table
- Wwe define load factor of the hash table

$$Load\ Factor = \alpha = \frac{N}{M}$$

  - With chaining, the load factor can be less, equal or larger than 1

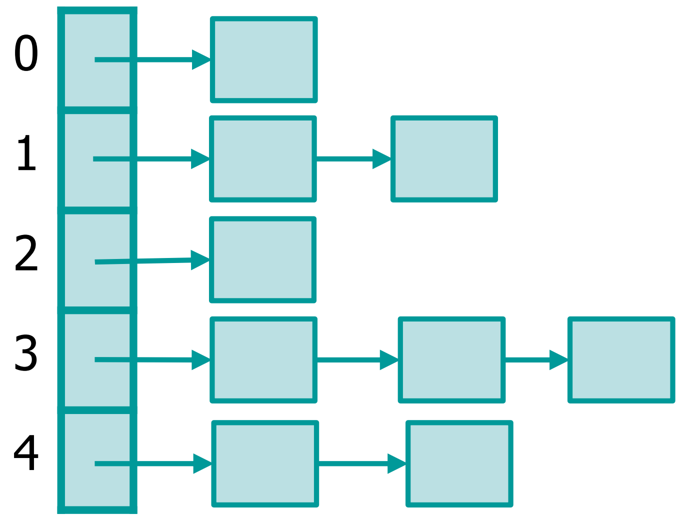


Complexity

- With unsorted lists and simple uniform hashing
  - $h(k)$ has the same probability to generate M output values
- Time cost $T(n)$

	Average Case	Worst Case
Insert	$O(1)$	
Search	$O(1+\alpha)$	$\theta(n)$
Delete	$O(1+\alpha)$	$\theta(n)$

In the worst case, the hash table degenerates into a list



Exemple

- Given the following set of keys (letters)

A S E R C H I N G X M P

- Insert them into a hash table of size

$$M = 5$$

- Using the module method for the hash function

$$h(k) = K \% M$$

  - Where k is the **positional order** of the key within the English alphabet (starting from **1**)

Solution

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

$$h(k) = k \;\%\; M$$
$$h(k) = k \;\%\; 5$$

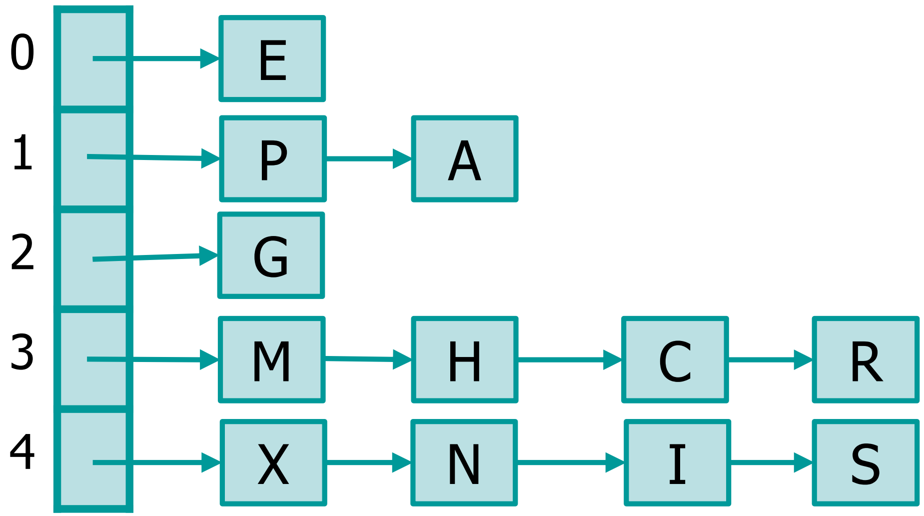
key	Order	h(k)
A	1	1
S	19	4
E	5	0
R	18	3
C	3	3
H	8	3
I	9	4

key	Order	h(k)
N	14	4
G	7	2
X	24	4
M	13	3
P	16	1

Solution

key	Order	h(k)
A	1	1
S	19	4
E	5	0
R	18	3
C	3	3
H	8	3
I	9	4

key	Order	h(k)
N	14	4
G	7	2
X	24	4
M	13	3
P	16	1



Method 2: Open Addressing

- Each cell of the table T stores a single element
  - All elements are stored in T
  - The load factor must always be less than 1

$$N \ll M \quad \rightarrow \quad Load\ Factor = \alpha = \frac{N}{M}$$

- When a collision occurs, it is necessary to look-for an empty cell
  - We generate a cell permutation, i.e., an order to search for an empty cell
  - We use the same order to insert and search the same key

Probing Functions

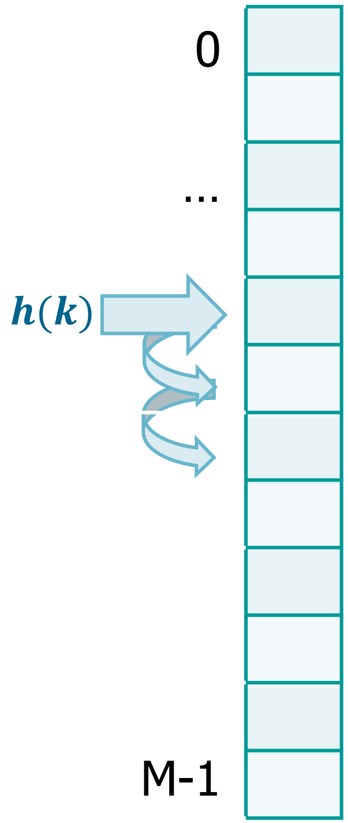
- We call the generation of the cell permutation probing
- There are several ways to perform probing
  - Linear probing
  - Quadratic probing
  - Double hashing
- A problem with open addressing is **clustering**
  - A cluster is a set of contiguous full cells which makes further collisions more probable in that area of the table

Linear Probing

- Given a key k

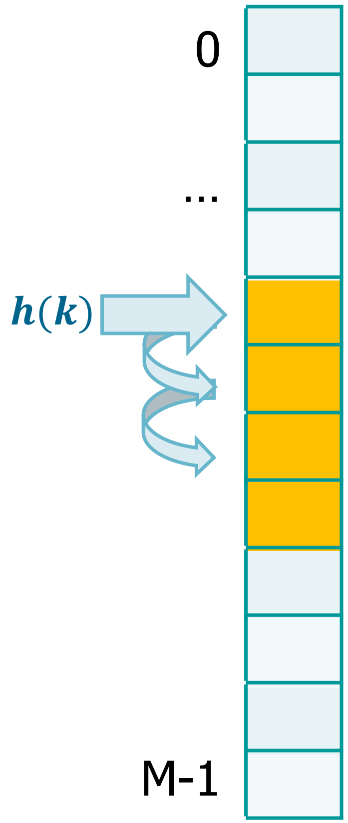
$$h'(k) = (h(k) + i) \,\%\, M$$

  - Variable i is the attempt counter
    - Start with $i = 0$ and increase it after every collision
- Algorithm
  - Set $i = 0$
  - Compute $h(k)$, then $h'(k)$
  - If the element is free, insert the key
  - Otherwise, increase i and repeat until an empty cell is found



Linear Probing

- Linear probing suffers from **primary clustering**
  - Long runs of occupied slots build up, increasing the average search time
  - Primary clusters are likely to arise
  - Runs of occupied slots tend to get longer
  - Unifor hashing is spoiled

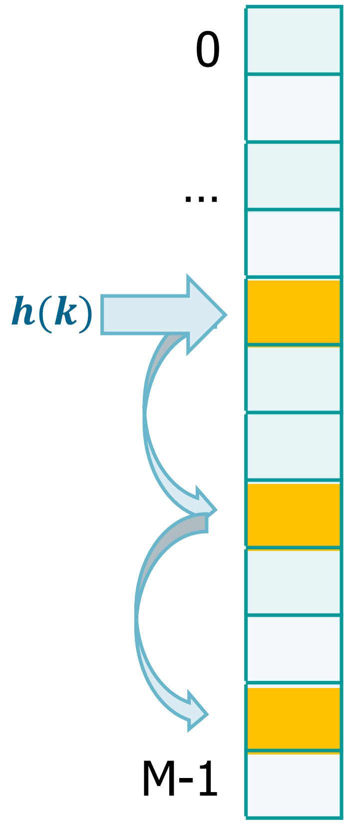


Quadratic Probing

- Given a key k

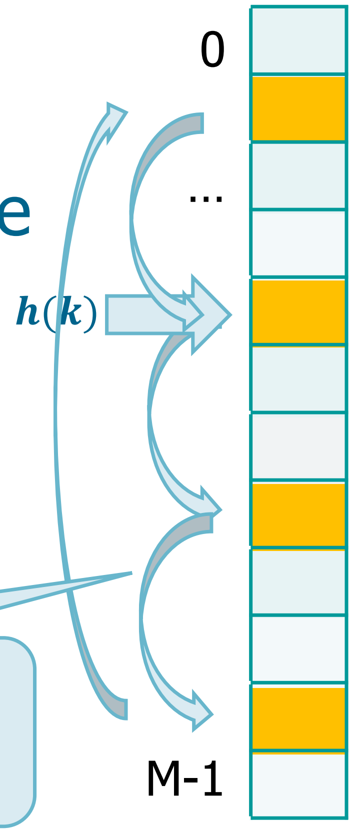
$$h'(k) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \ \% \ M$$

  - Variable i is the attempt counter
    - Start with $i = 0$ and increase it after every collision
- Algorithm
  - Set $i = 0$
  - Compute $h(k)$, then $h'(k)$
  - If the element is free, insert the key
  - Otherwise, increase i and repeat until an empty cell is found



Quadratic Probing

- In quadratic probing constants c_1 and c_2 must be selected carefully
  - They must guarantee that $h'(k)$ assumes distinct values for $1 \leq i \leq {}^{(M-1)}/_2$
  - If $M = 2 \cdot k$, we can select $c_1 = c_2 = {}^1/_2$ to generate all indexes between 0 and $M-1$
  - If M is prime and $\alpha < {}^1/_2$, we can select the following values
    - $c_1 = {}^1/_2$ and $c_2 = {}^1/_2$
    - $c_1 = 1$ and $c_2 = 1$
    - $c_1 = 0$ and $c_2 = 1$



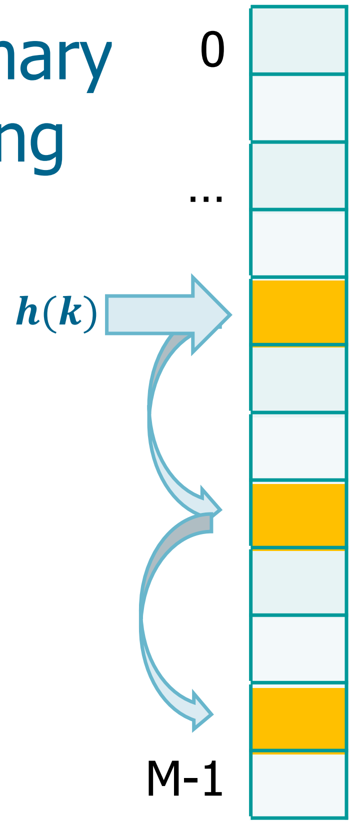
This condition must be avoided: The hash-table is partially empty but we scan the same elements over and over again

Quadratic Probing

- Quadratic probing suffers from **secondary clustering**

  logical continuity

  - A milder form of clustering where clustered elements are not contiguous
  - The same considerations made for the primary clustering hold also for this case of clustering

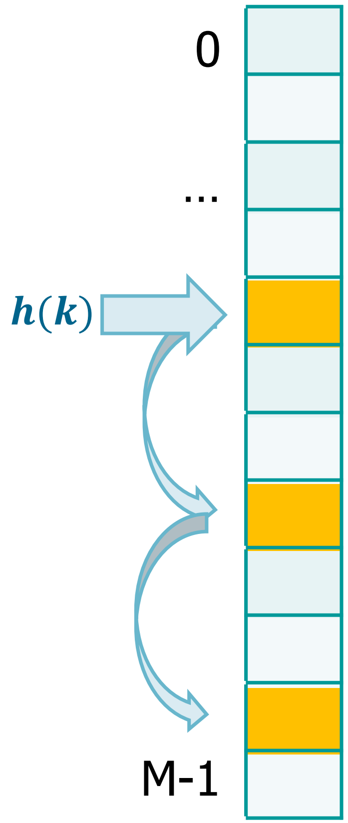


Double Hashing

- Given a key k

$$h'(k) = (h_1(k) + i \cdot h_2(k))\ \%\ M$$

  - Variable i is the attempt counter
    - Start with $i = 0$ and increase it after every collision
- Algorithm
  - Set $i = 0$
  - Compute $h_1(k)$, then $\mathrm{h}'(k)$
  - If the element is free, insert the key
  - Otherwise, increase i, compute $h_2(k)$, and repeat until an empty cell is found



Double Hashing

- In double hashing we must guarantee that the new value of $h'(k)$ differ from the previous one otherwise we enter an infinite loop
- To avoid this
  - h_2 should never return 0
  - $h_2 \ \% \ M$ should never return 0
- Examples

$$
\begin{aligned}
h_1(k) &= k \ \% \ M \qquad with\ M\ prime \\
h_2(k) &= 1 + k \ \% \ \widehat{M} \qquad with\ \widehat{M} = 97
\end{aligned}
$$

$h_2\ (k)$ never returns 0 and $h_2\ \%M$ never returns 0 if $M > 97$

Double Hashing

- Double hashing represents an improvement over linear or quadratic probing
  - As we vary the key, the initial probing position and the offset may vary **independently**
  - As a result, the performance of double hashing appears to be very close of the ideal scheme of uniform hashing

Probing and Delete

- With probing (all strategies) delete a key is a complex operation
  - Each delete operation potentially breaks a collision chain
  - For that reason open addressing is often used **only** when it is **not** necessary to delete keys
    - Hash tables limited to insertions and searches

Probing and Delete

- To extend the approach to hash tables with delete operations we must
  - Either substitute the deleted key with a sentinel key
    - The sentinel key is considered as
      - A full element during search operations and
      - An empty element during insertion operations
  - Or re-adjust clustered keys, to move some key into the deleted element

Example: Delete with Probing

- Delete E
  - We need to remind that keys E, S, R, and H collided into element 4

		Sentinel	Re-adjustment
0	A	A	A
1			
2	C	C	C
3			
4	E	**E**	H
5	S	S	S
6	R	R	R
7	H	H	
8			
9			
10			
11			
12			

Example

- Given the following set of keys (letters)

A S E R C H I N G X M P

- Insert them into a hash table of size

$$M = 13$$

- Using the module method with linear probing for the hash function

$$h(k) = ((k \% M) + i)\%M$$

  - Where k is the **positional order** of the key within the English alphabet (starting from **1**)

The constraint $\alpha < {}^1/_2$ is not respected

Solution

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

$$h(k) = k \;\%\; M = k \;\%\; 13$$
$$h'(k) = (k \;\%\; 13 + i) \;\%\; 13$$

key	Order	h(k)
A	1	1
S	19	6
E	5	5
R	18	5 → 6 → 7
C	3	3
H	8	8
I	9	9

key	Order	h(k)
N	14	1 → 2
G	7	7 → 8 → 9 → 10
X	24	11
M	13	0
P	16	3 → 4

Solution

Hash-table configuration after each insertion

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A					S						
	A				E	S						
	A				E	S	R					
	A		C		E	S	R					
	A		C		E	S	R	H				
	A		C		E	S	R	H	I			
	A	N	C		E	S	R	H	I			
	A	N	C		E	S	R	H	I	G		
	A	N	C		E	S	R	H	I	G	X	
M	A	N	C		E	S	R	H	I	G	X	
M	A	N	C	P	E	S	R	H	I	G	X	

Example

- Given the following set of keys (letters)

A S E R C H I N G X M P

- Insert them into a hash table of size

$$M = 13$$

- Using the module method with quadratic probing for the hash function

$$h(k) = \left((k \,\%\, M) + 0.5 \cdot i + 0.5 \cdot i^2\right) \%\, M$$

  - Where k is the **positional order** of the key within the English alphabet (starting from **1**)

The constraint $\alpha < {}^{1}/_{2}$ is not respected

Solution

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

$$h(k) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \ \% \ M$$
$$h(k) = (k \ \% \ M + 0.5 \cdot i + 0.5 \cdot i^2 \ ) \ \% \ 13$$

key	Order	h(k)
A	1	1
S	19	6
E	5	5
R	18	5 → 6 → 8
C	3	3
H	8	8 → 9
I	9	9 → 10

key	Order	h(k)
N	14	1 → 2
G	7	7
X	24	11
M	13	0
P	16	3 → 4

Solution

Hash-table configuration after each insertion

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A					S						
	A				E	S						
	A				E	S		R				
	A		C		E	S		R				
	A		C		E	S		R	H			
	A		C		E	S		R	H	I		
	A	N	C		E	S		R	H	I		
	A	N	C		E	S	G	R	H	I		
	A	N	C		E	S	G	R	H	I	X	
M	A	N	C		E	S	G	R	H	I	X	
M	A	N	C	P	E	S	G	R	H	I	X	

Example

- Given the following set of keys (letters)

  A S E R C H I N G X M P

- Insert them into a hash table of size

$$M = 13$$

- Using the module method with double hashing

$$h(k) = k \;\%\; M \quad and \quad h'(k) = 1 + k \;\%\; 97$$

  - Where k is the **positional order** of the key within the English alphabet (starting from **1**)

The constraint $\alpha < {}^{1}/_{2}$ is not respected

Solution

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

$$h(k) = \big(h(k) + i \cdot h'(k)\big) \ \% \ M$$
$$h(k) = (k \ \% \ 13 + \ i \cdot (1 + k \ \% \ 97)) \ \% \ 13$$

key	Order	h(k)
A	1	1
S	19	6
E	5	5
R	18	5 → 11
C	3	3
H	8	8
I	9	9

key	Order	h(k)
N	14	1 → 3 → 5 → 7
G	7	7 → 2
X	24	11 → 10
M	13	0
P	16	3→7→11→ 2→6→10→1→ 5→9→0→4

Solution

Hash-table configuration after each insertion

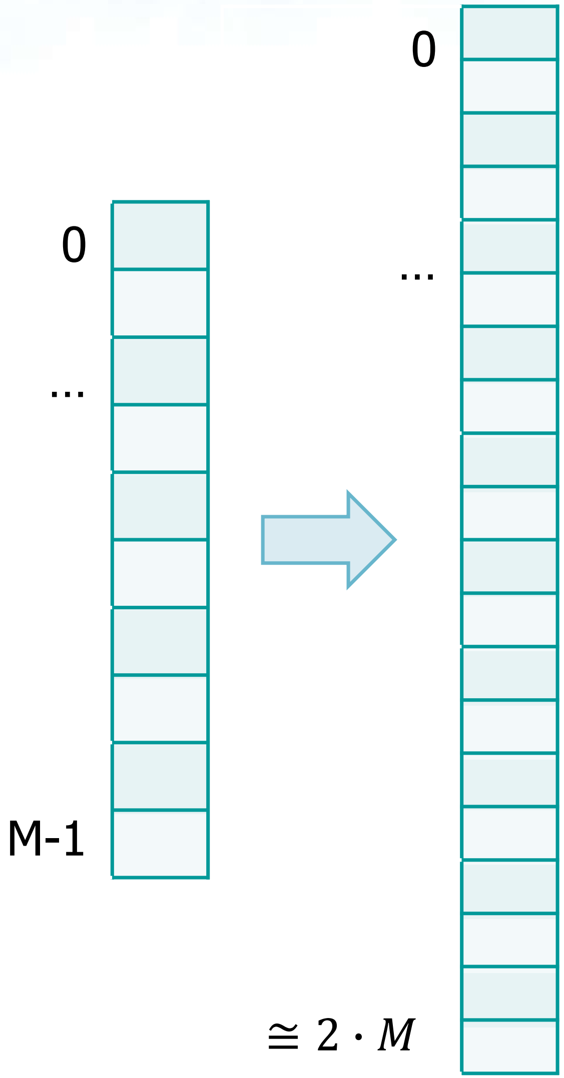
0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A					S						
	A				E	S						
	A				E	S					R	
	A		C		E	S					R	
	A		C		E	S		H			R	
	A		C		E	S		H	I		R	
	A		C		E	S	N	H	I		R	
	A	G	C		E	S	N	H	I		R	
	A	G	C		E	S	N	H	I	X	R	
M	A	G	C		E	S	N	H	I	X	R	
M	A	G	C	P	E	S	N	H	I	X	R	

Re-Hashing

- Hash tables offer exceptional performance when they are not overly full
  - The table size is the traditional dilemma of all array-based data structures
    - If we make the table too small, performance degrades and the table may overflow
    - If we make the table too big, memory gets wasted
- Rehashing or variable hashing attempts to circumvent this dilemma by expanding the hash table size whenever it gets too full

Re-Hashing

- Rehashing strategy
  - For every new entry into the map, check the load factor α
  - If, for example, $\alpha \geq 0.75$ then start **rehash**
    - For Rehashing, initialize a new tabler of a size about twice as large the previous one
    - Extract all elements from the original table and copy them into the new one



Final Considerations

- Hash Tables
  - Unique solution when keys do not have an ordering relation
  - Much faster on the average case
  - The hash table size must be **forecast** or the table may be **re-allocated**
- Trees (BST and variants)
  - Better worst-case performances when balanced trees are used
  - Easier to create with unknown or highly-variable number of keys
  - Allow operations on keys with an ordering relation