Symbol Tables

Hash Tables

Paolo Camurati and 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 a key
  - Reduce the storage requirements of direct-access tables from Θ(|U|) to Θ(|K|)
- Efficiency
  - Memory usage in the order of the number of keys stored in that table (not in the order of |U|)
    - M(K) = Θ(|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| << |U|

Definition

Previously **st**

Previously **getindex**

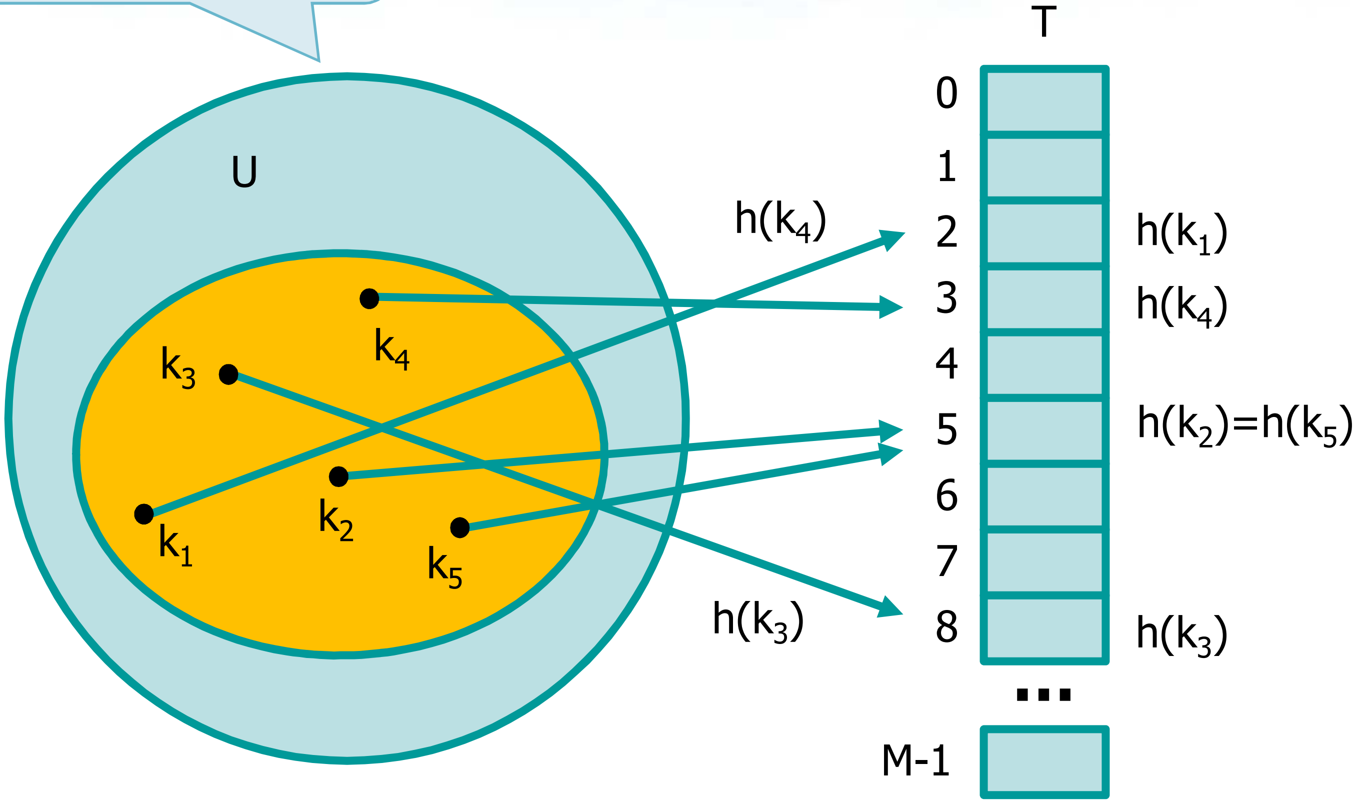
- It uses
  - A table (an array) to store the data
  - A function to transform each key into its position (index) into an array
- The table
  - Has size M and stores |K| elements
    - |K|<<|U|
  - Has addresses (indices) in the range [0, M-1]

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 h(k)
    - h: U → { 0, 1, ..., M-1 }
  - Each element of key k is stored at the address h(k)
  - As |K|<<|U| the hash function creates a mapping which is n:1, no more 1:1 as in the direct access tables

Hash Function

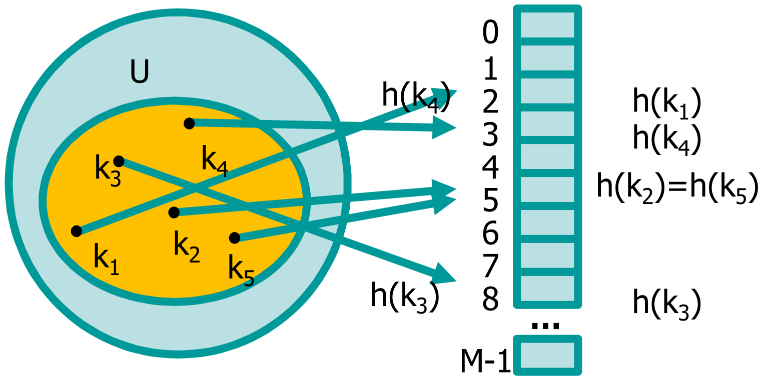
The mapping between $k \in U$ and elements in the table is |U|:M (not 1:1)



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**
- To **minimize** collisions we must **design** proper hash functions

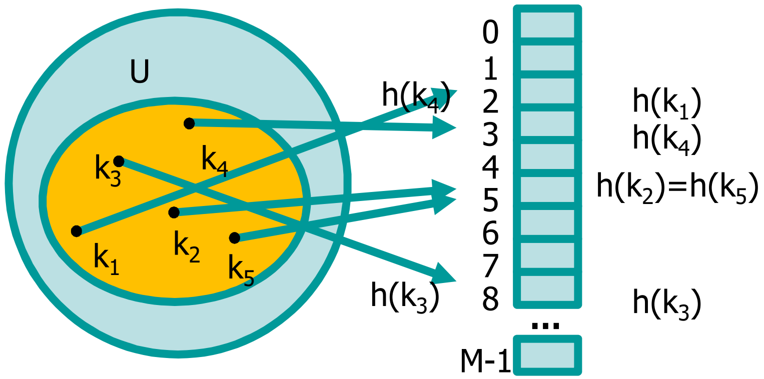
Problem # 1



Hash Function

- Collisions may always happen as the
  - Hash tables map |U| elements into |M| slots
    - The table cannot contain all keys within the universe of keys
  - The hash function cannot be perfect
    - The mapping may always create conflicts
  - We must understand **how** to deal with collisions

Problem # 2



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

The Multiplication Method

- If keys are floating point numbers
  - $k \in [s, t]$
  - $h(k) = \left\lfloor \frac{(k-s)}{(t-s)} \cdot M \right\rfloor$

Key’s range

⌊ ⌋ = floor =
largest integer smaller than

- Example
  - M = 97
  - $k \in [0, 1.0] = 0.513871$
  - $h(k) = \left\lfloor \frac{(0.513871-0)}{(1-0)} \cdot 97 \right\rfloor = 49$

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

The Module Method

- If keys are integer numbers
  - $h(k) = k \% M$
  - Alternative function
    - $h(k) = 1 + k \% M'$, with $M'<M$
- 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
  - If $M = 10^n$ we use only the last n decimal digits
  - Keys will not evenly distribute

Fast and easy to compute

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

The Module Method

- Examples
  - M = 19
    - k = 11 → h(k) = 11 % 19 = 11
    - k = 31 → h(k) = 31 % 19 = 12
    - k = 29 → h(k) = 29 % 19 = 10

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

The Multiplication-Module Method

- If keys are integer numbers
  - Given a constant value A, the hash function can be computed as
    - $A \in \,]0, 1[$
    - $h(k) = \lfloor k \cdot A \rfloor \% M$
  - A good value for A is
    - $A = \frac{(\sqrt{5}-1)}{2} = 0.6180339887$

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

The Multiplication-Module Method

- Examples
  - M = 19
    - $k = 11 \rightarrow h(k) = \lfloor 11 \cdot A \rfloor \% 19 = 6 \% 19 = 6$
    - $k = 31 \rightarrow h(k) = \lfloor 11 \cdot A \rfloor \% 19 = 19 \% 19 = 9$

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

The Modular Method

- If keys 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
    - The result is to transform the string into an integer
    - Once the integer is obtained the module method can be applied
      - h(k) = k % M

The Modular Method

- Example
  - K = "now"
  - M = 19
    - $h(k) = p_n \cdot b^n + p_6 \cdot b^6 + \ldots + p_2 \cdot b^2 + p_1 \cdot b^1 + p_0 \cdot b^0$

$$= ('n' \cdot 128^2 + 'o' \cdot 128^1 + 'w') \ \% \ 19$$

$$= (110 \cdot 128^2 + 111 \cdot 128^1 + 119) \ \% \ 19$$

$$= 1816567 \ \% \ 19$$

$$= 15$$

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

To each character we may use the corresponding ASCII value

The Modular Method

- If keys are **long** alphanumeric strings
  - The previous computation overflows, and the result cannot be represented on a reasonable number of bits
  - In this case, it is possible to use the Horner's method to rule-out M multiples after each step, instead of doing that after the application of the modular technique
    - $h(k) = p_{n-1} \cdot b^{n-1} + p_{n-2} \cdot b^{n-2} + \ldots + p_2 \cdot b^2 + p_1 \cdot b^1 + p_0 \cdot b^0$

      $= (((((p_n \cdot b + p_{n-1}) \cdot b + p_{n-2}) \cdot b + \ldots + p_2) \cdot b + p_1) \cdot b + p_0$

      $= ((((((p_{n-1} \% M) \cdot b + p_{n-2}) \% M) \cdot b + p_{n-3}) \cdot b) \% M \ldots$

The Modular Method

- The resulting implementation is the following one
    - Base b=128

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

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

  return h;
}
```

The Modular Method

- Example
  - k = "averylongkey"
  - b = 128
  - h(k) =

$$= 97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^{9} + 114 \cdot 128^{8}$$

$$+ 121 \cdot 128^{7} + 108 \cdot 128^{6} + 111 \cdot 128^{5} + 110 \cdot 128^{4}$$

$$+ 103 \cdot 128^{3} + 107 \cdot 128^{2} + 101 \cdot 128^{1} + 121 \cdot 128^{0}$$

$$= (((((((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114)$$

$$\cdot 128 + 121) \cdot 128 + 108) \cdot 128 + 111) \cdot 128 + 110)$$

$$\cdot 128 + 103) \cdot 128 + 107) \cdot 128 + 101) \cdot 128 + 121$$

$$= (((((((((((97 \% M) \cdot 128 + 118) \% M) \cdot 128 + 114) \% M)$$

$$\cdot 128 + 121) \% \ldots$$

The Modular Method

- To obtain a uniform distribution we must have a collision probability for 2 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;
}
```

The Modular Method

- 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;
}
```

Collisions

- A collision happens when
  - $k_i \neq k_j \rightarrow h(k_i)=h(k_j)$
- Collisions are inevitable, as
  - $|K| \sim |M| << |U|$
  - Hash functions are not perfect and do not distribute keys uniformly
- Then, it is necessary to
  - Minimize their number
    - Select a good hash function for each specific problems / set of keys
  - Deal with collisions when they occur

Collisions

- Collisions can be dealt with
  - Linear chaining
    - For each hash table entry, a list of elements stores all data items having the same hash function value
  - Open addressing
    - For each collision, it tries to place the same element somewhere else (in another table entry) within the table

Linear Chaining

- More elements can be stored in the same table location
  - A table element does not contain a key anymore
  - Each element points to a linked list
- Insert an element implies inserting it on the list
  - The most efficient way is to do this operation on the list head
- Delete an element implies
  - A list search
    - Lists are not usually sorted as insertions are on the head
  - A delete operation from the list

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
    - Lists too long 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)

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 = 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

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

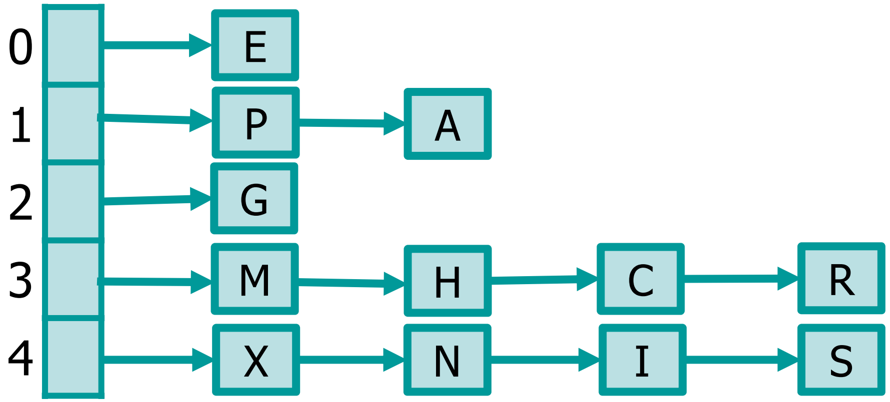
key	Order	h(k)
A	1	1
S	19	4
E	5	0
R	18	3
C	3	3
H	8	3
I	9	4

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



Complexity

- With non-ordered lists
  - N = number of stored elements
    - It should be of the same order of |K|
  - M = size of the hash table
- Simple Uniform Hashing
  - h(k) has the same probability to generate M output values
- Definition
  - Load factor = $\alpha = \frac{N}{M}$
  - It can be less, equal or larger than 1

Complexity

- Insert
  - $T(n) = O(1)$
- Search
  - Worst case
    - $T(n) = \Theta(N)$
  - Average case
    - $T(n) = O(1+\alpha)$
- Delete
  - As the search

Open Addressing

- Each cell table T can store a single element
- All elements are stored in T
- Once there is a collision it is necessary to look-for an empty cell with **probing**
  - Generate a cell permutation, i.e, an order to search for an empty cell
  - The same order has to be used to insert and to search a key

$N \leq M$
$\alpha \leq 1$

Probing Functions

- 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 = 2K, select $c_1 = c_2 = ½$ to generate all indexes between 0 and M-1
  - If M is prime and $\alpha < ½$ the following values
    - $c_1 = ½$ and $c_2 = ½$
    - $c_1 = 1$ and $c_2 = 1$
    - $c_1 = 0$ and $c_2 = 1$

Quadratic Probing

- Quadratic probing suffers from **secondary clustering**
  - 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 $h'(k)$
  - 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 guarantte 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
  - $h_1(k) = k \% M$ and M prime
  - $h_2(k) = 1 + k \% 97$
  - $h_2(k)$ 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

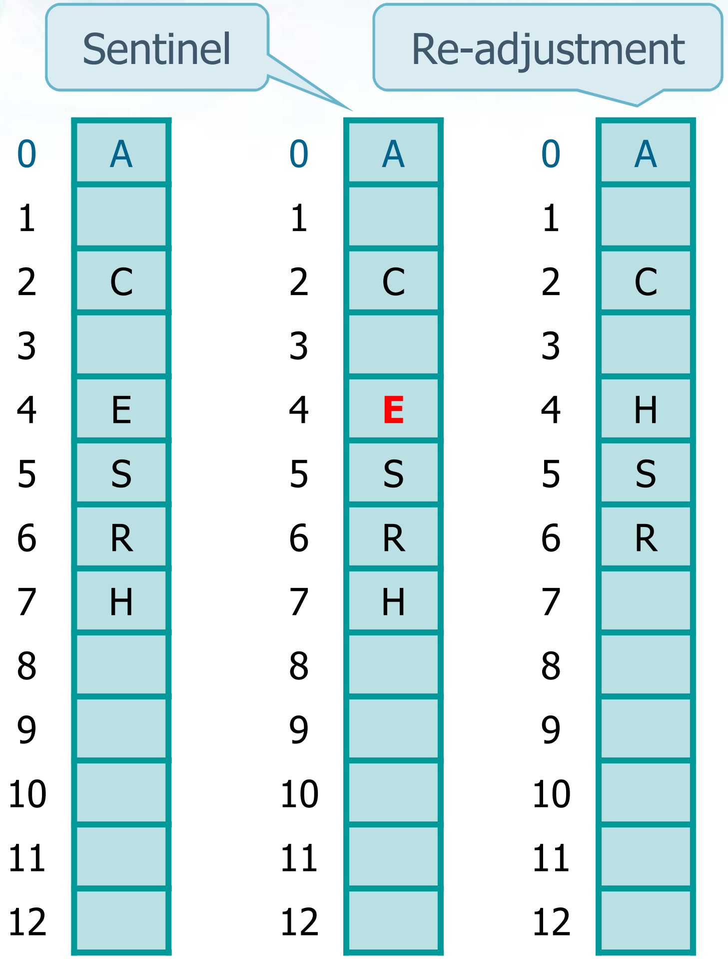
- With probing (all strategies) delete a key is a complex operation
  - Each delete operation potentially breaks a collision chain
  - For that reason open addressing is often used **only** when it is **not** necessary to delete keys
    - Hash tables limited to insertions and searches

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 for the hash function with linear probing
  - h(k) = k % M
  - Where k is the positional order of the key within the English alphabet

The constraint
$\alpha < \frac{1}{2}$
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 for the hash function with quadratic probing
  - h(k) = k % M
  - Where k is the positional order of the key within the English alphabet

The constraint
$\alpha < \frac{1}{2}$
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

$$
\begin{aligned}
h(k) &= (h(k) + i \cdot h(k)) \ \% \ M \\
&= (k \ \% \ M + 0.5 \cdot i + 0.5 \cdot i^2) \ \% \ 13
\end{aligned}
$$

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 L
- Insert them into a hash table of size
  - M = 13
- Using the module method for the hash function with double hashing
  - h'(k) = k % M
  - h"(k) = 1 + k % 97
  - Where k is the positional order of the key within the English alphabet

The constraint
$\alpha < \frac{1}{2}$
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

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

$$= (k \% M + i \cdot (k \% 97)) \% 13$$

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	

Comparison

- Hash Table
  - Unique solution when keys do not have an ordering relation
  - Much faster on the average case
  - The hast table size mut be forecast or it 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